Using Fast Matrix Multiplication to Find Basic Solutions[*]

Peter A. Beling[†] and Nimrod Megiddo[‡]

Februaray 1993

Abstract. We consider the problem of finding a basic solution to a system of linear constraints (in standard form) given a non-basic solution to the system. We show that the known arithmetic complexity bounds for this problem admit considerable improvement. Our technique, which is similar in spirit to that used by Vaidya to find the best complexity bounds for linear programming, is based on reducing much of the computation involved to matrix multiplication. Consequently, our complexity bounds in their most general form are a function of the complexity of matrix multiplication. Using the best known algorithm for matrix multiplication, we achieve a running time of $O(m^{1.62}n)$ arithmetic operations for an $m \times n$ problem in standard form. Previously, the best bound was $O(m^2n)$ arithmetic operations.

Key Words: Computational complexity, linear programming, basic solutions, interior-point methods, fast matrix multiplication.

1. Introduction

Consider the standard-form system of linear constraints

$$\begin{aligned} \boldsymbol{Ax} &= \boldsymbol{b} \\ \boldsymbol{x} &\geq \boldsymbol{0}\,, \end{aligned}$$

where $\boldsymbol{A} \in \Re^{m\times n}$ is assumed to have linearly independent rows, $\boldsymbol{b} \in \Re^m$, and $\boldsymbol{x} \in \Re^n$. A solution $\boldsymbol{x}$ of this system is said to be *basic* if the set of columns $\boldsymbol{A}_{\bullet j}$ with $x_j \neq 0$ is linearly independent. Thus, a basic solution has at most m positive components.

[*]Research supported in part by ONR contract N00014-91-C-0026.

[†]Department of Industrial Engineering and Operations Research, University of California, Berkeley, CA 94720, and IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099

[‡]IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099, and School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel.

The problem of finding a basic solution given a non-basic one to arises frequently in linear programming, especially in the context of interior-point methods. For simplicity, we call this problem *basis crashing.*

We are interested in the arithmetic complexity of basis crashing, *i.e.*, the number of elementary arithmetic operations needed to solve the problem as a function of its dimension. (See, *e.g.*, [8] for detailed material on arithmetic complexity in general.) Previously, the best arithmetic complexity bound known for the $m \times n$ basis crashing problem was $O(m^2 n)$ arithmetic operations. In this note we show that this bound admits considerable improvement. Our technique, which is similar in spirit to that used by Vaidya [7] to improve the complexity bounds for linear programming, is based on reducing much of the computation in basis crashing to matrix multiplication. Consequently, our complexity bounds in their most general form are a function of the complexity of matrix multiplication.

Denote by $T(k)$ the number of arithmetic operations required to multiply two $k \times k$ matrices. We show that the $m \times n$ problem can be solved in $O(m^{\frac{3-\delta}{2-\delta}} n)$ arithmetic operations, where δ is any scalar known to satisfy $T(k) = O(k^{2+\delta})$. Using the best known algorithm for matrix multiplication, we achieve a running time of $O(m^{1.62} n)$ arithmetic operations for the $m \times n$ basis crashing problem.

In the remainder of the paper we adopt the following notation and terminology. Matrices and vectors are denoted bold-faced. Ordinary capital letters are often used to denote ordered sequences of indices with respect to matrices or vectors. Given a sequence $C = (j_1, \ldots, j_k)$, we use $\boldsymbol{A}_C$ to denote submatrix $[\boldsymbol{A}_{\bullet j_1} \cdots \boldsymbol{A}_{\bullet j_k}]$, where $\boldsymbol{A}_{\bullet j}$ is the j-th column of $\boldsymbol{A}$. Likewise, we use $\boldsymbol{x}_C$ to denote the restriction of the vector $\boldsymbol{x}$ to the indices in C. A sequence B is a *basic sequence* with respect to the matrix $\boldsymbol{A}$ if the submatrix $\boldsymbol{A}_B$ is (square and) nonsingular. This last matrix is called the *basis* corresponding to the sequence B. Additionally, the matrix $\boldsymbol{A}$ is said to be in *canonical form* with respect to the basic sequence B if $\boldsymbol{A}_B$ is the identity matrix.

2. Standard algorithms

In this section we give a brief review of the theory and implementation of the existing basis crashing algorithms.

The basis crashing problem can be viewed in geometric terms as the problem of finding the vertex of the polyhedron $P = \{\boldsymbol{x} \mid \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b},\ \boldsymbol{x} \geq \boldsymbol{0}\}$ given any point in P. This problem admits a trivial recursive solution: Move in any bounded direction until hitting a face of the positive orthant. The resulting point belongs to a polyhedron that can be described in a lower dimension. This algorithm is equally simple algebraically.

The algebraic observation is that if the columns of $\boldsymbol{A}$ that correspond to positive component of a given solution $\bar{\boldsymbol{x}}$ are linearly dependent, then we can find a vector $\boldsymbol{z} \neq \boldsymbol{0}$

such that $\boldsymbol{A}\boldsymbol{z} = \boldsymbol{0}$, and among the components of $\boldsymbol{z}$, only the ones corresponding to these columns may be nonzero. It follows that we can also find a scalar θ such that the vector $\hat{\boldsymbol{x}} = \bar{\boldsymbol{x}} + \theta\boldsymbol{z}$ is a solution to the given system and has at least one less positive component than the given solution $\bar{\boldsymbol{x}}$. The columns $\boldsymbol{A}_{\bullet j}$ such that $\hat{\boldsymbol{x}} = \boldsymbol{0}$ can then be eliminated from further consideration, yielding a smaller problem. Successive applications of this procedure yield a basic solution as desired.

The algorithm outlined above is conveniently described and implemented using basic sequences and canonical forms. Knowing the canonical form of the matrix $\boldsymbol{A}$ with respect to a basic sequence makes it particularly easy to find a nonzero vector in the null space of $\boldsymbol{A}$, and hence a solution to the given system that has at one less positive component than the given solution. Consider first an algorithm that solves the basis crashing problem under the assumption that an initial basic sequence and basis inverse for the coefficient matrix are given as input. The algorithm admits a recursive description.

Algorithm 2.1. Given are the following: $\boldsymbol{A} \in \Re^{m\times n}$ with linearly independent rows, $\boldsymbol{b} \in \Re^m$, a solution vector $\bar{\boldsymbol{x}} \geq \boldsymbol{0}$ such that $\boldsymbol{A}\bar{\boldsymbol{x}} = \boldsymbol{b}$, and a basic sequence B for $\boldsymbol{A}$.

Step 1. Select an index $k \notin B$ with $\bar{x}_k > 0$.

Step 2. Compute the vector $\bar{\boldsymbol{A}}_{\bullet k} = \boldsymbol{A}_B^{-1}\boldsymbol{A}_{\bullet k}$. Let $\bar{A}_{jk}$ denote the j-th entry of $\bar{\boldsymbol{A}}_{\bullet k}$. There are two cases to consider:

(i) $\bar{\boldsymbol{A}}_{\bullet k} \geq \boldsymbol{0}$. Set $\theta = \bar{x}_k$ and $\hat{B} = B$.

(ii) $\bar{\boldsymbol{A}}_{\bullet k} \not\geq \boldsymbol{0}$. Set $\theta = \min\{\phi, \bar{\boldsymbol{x}}_k\}$, where $\phi = \min\{\bar{x}_j/|\bar{A}_{jk}| \,:\, \bar{A}_{jk} < 0\}$. Set $\hat{B} = B \cup \{k\} \setminus \{\ell\}$, where $\ell = \arg\min\{\bar{x}_{B(j)}/|\bar{A}_{jk}| \,:\, \bar{A}_{jk} < 0\}$.

Step 3. Form the n-vector $\hat{\boldsymbol{x}}$ as follows:

$$\hat{\boldsymbol{x}}_j = \begin{cases} \bar{\boldsymbol{x}}_j + \theta\bar{A}_{jk} & \text{if } j \in B \\ \bar{\boldsymbol{x}}_k - \theta & \text{if } j = k \\ \bar{\boldsymbol{x}}_j & \text{otherwise.} \end{cases}$$

Step 4. Stop if $\hat{\boldsymbol{x}}$ has at most m positive components. Otherwise continue recursively after updating the input with the solution $\hat{\boldsymbol{x}}$ and the basic sequence $\hat{B}$.

To establish the validity of the algorithm's recursive description, it suffices to show that $\hat{B}$ is a basic sequence for the matrix $\boldsymbol{A}$ and that $\hat{\boldsymbol{x}}$ is nonnegative and satisfies $\boldsymbol{A}\hat{\boldsymbol{x}} = \boldsymbol{b}$. That $\hat{B}$ is a basic sequence for $\boldsymbol{A}$ follows immediately from the fact that $\bar{A}_{\ell k}$ is necessarily nonzero in case (ii) (there is nothing to prove in case (i)). That $\hat{\boldsymbol{x}}$ satisfies $\boldsymbol{A}\hat{\boldsymbol{x}} = \boldsymbol{b}$ can be seen from the expansion

$$\begin{aligned} \boldsymbol{A}(\bar{\boldsymbol{x}} - \hat{\boldsymbol{x}}) &= -\theta\sum_{j\in B}\boldsymbol{A}_{\bullet j}\bar{A}_{jk} + \theta\boldsymbol{A}_{\bullet k} = \theta(-\boldsymbol{A}_B\bar{A}_{jk} + \boldsymbol{A}_{\bullet k}) \\ &= \theta(-\boldsymbol{A}_B\boldsymbol{A}_B^{-1}\boldsymbol{A}_{\bullet k} + \boldsymbol{A}_{\bullet k}) = \boldsymbol{0} \end{aligned}$$

Likewise, it straightforward to verify that the rules defining θ ensure that $\hat{\boldsymbol{x}}$ is nonnegative.

The total number of (recursive) iterations performed by the algorithm is at most $n-m$. To see this, note that at the start of each iteration a single non-basic column with positive coefficient is selected for consideration. During the course of the iteration, either the selected column is entered into the basis or the corresponding coefficient is reduced to zero (or both). The coefficients of all other non-basic columns remain constant, and the coefficient of any column which leaves the basis is necessarily zero. It follows that any column which has previously been selected for consideration or that has previously been basic is either basic or non-basic with coefficient zero through the remainder of the computation. Such columns are effectively dropped from the problem.

The computational effort in Algorithm 2.1 clearly centers around the construction of the canonical column $\bar{\boldsymbol{A}}_{\bullet k} = \boldsymbol{A}_B^{-1}\boldsymbol{A}_{\bullet k}$. There are at least two reasonable ways of generating this columns in practice. The more familiar, perhaps, of these is a tableau-based pivoting procedure similar to that of the simplex method. One begins with the canonical form of $\boldsymbol{A}$ with respect to the initial sequence B, and uses Gauss-Jordan pivots to explicitly maintain canonical form as the basic sequences change and as columns are dropped from further consideration.

One can also envision an implicit scheme similar to the revised simplex method. Here it is the inverse of the basis and not canonical form that is maintained at each iteration. The canonical column is generated by multiplying the basis inverse by the appropriate column from the original coefficient matrix. The basis inverse itself can be maintained explicitly through Gaussian elimination or implicitly as a product of elementary pivot matrices. In the latter case it suffices to store only the nonzero column of each pivot matrix.

It easy to see that these approaches are essentially the same, and that they can be implemented in $O(m^2n)$ arithmetic operations. Moreover, the initialization step of finding a basic sequence B for $\boldsymbol{A}$ and computing $\boldsymbol{A}_B^{-1}$ can be done in $O(m^2n)$ time by (say) using Gauss-Jordan pivots to reduce the left-hand side of the matrix $[\boldsymbol{A}\ \boldsymbol{I}]$ (cf. [4]). Hence, we have the following bound on the complexity of the general problem:

Proposition 2.2. *The $m \times n$ basis crashing problem can be solved in $O(m^2n)$ arithmetic operations.*

3. Improvements using fast matrix multiplication

In this section we show that the asymptotic complexity bound for basis crashing given in the last section admits a considerable improvement. Our technique is based on reducing much of the computation to matrix multiplication, and consequently our complexity bounds in their most general form are a function of the complexity of matrix multiplication. Using the best known algorithm for matrix multiplication, we achieve a running time of $O(m^{1.62}n)$ arithmetic operations.

During each iteration of the new algorithm, we work with a given basis and a small number of additional non-basic columns. These non-basic columns (and no others) are first brought into canonical form with respect to the basis. The resulting subproblem is then solved and a new basic sequence identified using the procedures described in the last section. The main computational work in the algorithm is divided between that used to bring each new set of columns into canonical form and that used to maintain the canonical form as each of these columns is either added to the basis or dropped from further consideration. Indeed, the number of columns considered in each iteration, and hence the total number of iterations performed, will be chosen to balance the complexity of these two tasks.

We consider first an algorithm that solves the basis crashing problem under the assumption that an initial basic sequence and basis inverse for the coefficient matrix are given as input. Later we show that these initial objects can be found using much the same procedure. The algorithm admits a recursive description.

Algorithm 3.1. Given are the following: $\boldsymbol{A} \in \Re^{m\times n}$ with linearly independent rows, $\boldsymbol{b} \in \Re^m$, a solution vector $\bar{\boldsymbol{x}} \geq \boldsymbol{0}$ such that $\boldsymbol{A}\bar{\boldsymbol{x}} = \boldsymbol{b}$, a basic sequence B for $\boldsymbol{A}$ and the associated basis inverse $\boldsymbol{A}_B^{-1}$, and an integer r.

Step 1. Select a sequence N of length r of indices $j \notin B$ with $\bar{x}_j > 0$. Form the product $\boldsymbol{A}_B^{-1}\boldsymbol{A}_N$.

Step 2. Let $C = B \cup N$. Using Algorithm 2.1, find a basic solution to the system $\boldsymbol{A}_B^{-1}\boldsymbol{A}_C\boldsymbol{z} = \boldsymbol{A}_B^{-1}\boldsymbol{A}\bar{\boldsymbol{x}}_C$, taking $\bar{\boldsymbol{z}} = \bar{\boldsymbol{x}}_C$ as the initial solution, B as the initial basic sequence, and $\boldsymbol{A}_B^{-1}$ as the initial basis inverse. Let $\hat{\boldsymbol{z}}$ denote the basic solution obtained in this way, and let $\hat{B}$ denote the corresponding basic sequence. Set

$$\hat{\boldsymbol{x}}_j = \begin{cases} \hat{\boldsymbol{z}}_j & \text{if } j \in C \\ \bar{\boldsymbol{x}}_j & \text{otherwise.} \end{cases}$$

Stop if $\hat{\boldsymbol{x}}$ has at most m positive components.

Step 3. Calculate $\boldsymbol{A}_{\hat{B}}^{-1}$.

Step 4. Continue recursively after updating the input with the solution $\hat{\boldsymbol{x}}$, the basic sequence $\hat{B}$, and the basis inverse $\boldsymbol{A}_{\hat{B}}^{-1}$.

Note that the solution vector produced during each (recursive) iteration has at least r fewer positive components than that of the previous iteration. Hence, the algorithm executes each step at most $\lceil \frac{n-m}{r} \rceil$ times.

We now turn to an analysis of the complexity of Algorithm 3.1 Ultimately, we shall express this complexity solely in terms of the problem dimensions m and n. For the moment, however, we allow additional dependence on the number of non-basic columns considered in each iteration (the parameter r) and on the cost of matrix multiplication. Later we shall choose a value for r which minimizes the complexity of the algorithm as a function of m, n, and the cost of matrix multiplication. We then make use of some well-known algorithms to bound the cost of matrix multiplication. Recall that $T(r)$ denotes the number of arithmetic operations required to multiply two $r \times r$ matrices.

Proposition 3.2. *Given an initial basic sequence and basis inverse, Algorithm 3.1 can be used to solve the $m \times n$ basis crashing problem in $O\left(\left(mr + \frac{m^2 T(r)}{r^3}\right)(n-m)\right)$ arithmetic operations.*

Before proving Proposition 3.2, we first recall two known facts concerning the cost of matrix multiplication. Denote by $V(k)$ the number of arithmetic operations required to invert a $k \times k$ matrix.

Fact 3.3. $T(k) = \Omega(k^2)$.

Fact 3.4. $T(k) = \theta(V(k))$, *i.e.*, $T(k) = O(V(k))$ *and* $V(k) = O(T(k))$.

See, *e.g.*, [1] for proof of Fact 3.3 and [6] for proof of Fact 3.4.

Proof of Proposition 3.2: The dominant effort in step 1 of the algorithm is the multiplication of the $m \times m$ matrix $\boldsymbol{A}_B^{-1}$ by the $m \times r$ matrix $\boldsymbol{A}_N$. We may assume, without loss of generality, that r divides m, since otherwise the matrices can be padded suitably with zeroes. Partitioning each of the matrices into blocks of size $r \times r$, we can create a new multiplication problem of dimension $\frac{m}{r} \times \frac{m}{r}$ and $\frac{m}{r} \times 1$ in which each 'element' is a block. This multiplication can be done using the ordinary row by column procedure with $\frac{m^2}{r^2}$ multiplications of blocks and $\left(\frac{m}{r}\right)\left(\frac{m}{r} - 1\right)$ additions of these products. Using this procedure, the complexity of the step is $O\left(\frac{m^2}{r^2} T(r)\right)$.

The dominant effort in step 2 is the solution of an $m \times (m+r)$ basis crashing problem in which an initial basic sequence and basis inverse are given. From the discussion in section 2, this can be done in $O(mr^2)$ arithmetic operations.

Step 3 consists of the computation of $\boldsymbol{A}_{\hat{B}}^{-1}$. From the description of the algorithm, we see that $\boldsymbol{A}_{\hat{B}}$ differs from $\boldsymbol{A}_B$ in ℓ columns, for some $\ell \leq r$. In particular, we can write

$$\boldsymbol{A}_{\hat{B}} = \boldsymbol{A}_B + \boldsymbol{U}\boldsymbol{V}^T,$$

where $\boldsymbol{U}$ is an $m \times \ell$ matrix consisting of the nonzero columns of $\boldsymbol{A}_{\hat{B}} - \boldsymbol{A}_B$, and $\boldsymbol{V}$ is an $m \times \ell$ matrix consisting of appropriate columns of the identity matrix. Since $\boldsymbol{A}_{\hat{B}}$

is a rank ℓ perturbation of $\boldsymbol{A}_B$, the well-known Sherman-Morrison-Woodbury formula (see, *e.g.*, [3]), gives a closed-form expression for $\boldsymbol{A}_{\hat{B}}^{-1}$ in terms of $\boldsymbol{A}_B^{-1}$. In particular,

$$\boldsymbol{A}_{\hat{B}}^{-1} = \boldsymbol{A}_B^{-1} - \boldsymbol{A}_B^{-1}\boldsymbol{U}(\boldsymbol{I} + \boldsymbol{V}^T\boldsymbol{A}_B^{-1}\boldsymbol{U})^{-1}\boldsymbol{V}^T\boldsymbol{A}_B^{-1},$$

where $\boldsymbol{I}$ is the $\ell \times \ell$ identity matrix. It is easy to verify that, since $\boldsymbol{A}_B^{-1}$ is known, this formula can be evaluated using a constant number of the following operations: (i) multiplication of an $m \times m$ matrix by an $m \times \ell$ matrix (and the transpose of this operation), (ii) multiplication of an $m \times \ell$ matrix by an $\ell \times m$ matrix, (iii) multiplication of an $\ell \times m$ matrix by an $\ell \times m$ matrix, (iv) inversion of an $\ell \times \ell$ matrix, (v) addition of $\ell \times \ell$ matrices, and (vi) addition of $m \times m$ matrices.

Obviously, the additions can be done in $O(m^2)$ arithmetic operations. Likewise, by Fact 3.4, the inversions can be done in $O(T(r))$ arithmetic operations. We can perform the multiplications using $\ell \times \ell$ blocks in the same (or a very similar) manner as that described in the analysis of step 1. The dominant multiplication term is then $O\left(\frac{m^2T(r)}{r^2}\right)$. Hence, the complexity of step 3 is $O\left(\frac{m^2T(r)}{r^2}\right)$.

The proof of the proposition follows immediately by combining the bounds on the complexity of each step given above and noting that, given an initial basic sequence and basis inverse, the algorithm executes each step at most $\lceil\frac{n-m}{r}\rceil$ times. ∎

We now consider the initialization problem of finding a basis for an arbitrary $m \times n$ matrix. Taking a clue from the initialization algorithms developed for the simplex method, it is not surprising that this problem can be solved by applying a slight variant of the main algorithm (Algorithm 3.1) to an artificial problem. In particular, given a matrix $\boldsymbol{A}$, we form the artificial matrix $[\boldsymbol{A}\ \boldsymbol{I}]$. Beginning with the trivial basis composed of the artificial columns, we maintain a basis for the artificial matrix. At each subsequent iteration we attempt to increase the number of non-artificial columns in the basis by bringing r non-basic and non-artificial columns into canonical form and then using Gauss-Jordan reduction to pivot these columns into the basis. If an appropriate pivot selection rule is followed, the final basis will contain no artificial columns.

Algorithm 3.5. Given are the following: $\boldsymbol{A} \in \Re^{m\times n}$ with linearly independent rows, a basic sequence B for $[\boldsymbol{A}\ \boldsymbol{I}]$ (where $\boldsymbol{I}$ is the $m \times m$ identity matrix), the associated basis inverse $([\boldsymbol{A}\ \boldsymbol{I}])_B^{-1}$, an integer r, and a set of 'previously considered' indices $\mathcal{C}$.

Step 0. Label the indices $n+1, \ldots, n+m$ 'artificial'. Set $\mathcal{C} = \emptyset$.

Step 1. Select a sequence N of length r from those indices that are not artificial and are not in $B \cup \mathcal{C}$. Form the set $\hat{\mathcal{C}} = \mathcal{C} \cup N$ and the matrix $\bar{\boldsymbol{A}}_N = ([\boldsymbol{A}\ \boldsymbol{I}])_B^{-1}\boldsymbol{A}_N$.

Step 2. Apply the Gauss-Jordan pivoting procedure (see, *e.g.*, [4]) to reduce the left-hand side of the matrix $[\bar{\boldsymbol{A}}_N\ \boldsymbol{I}]$, updating the basic sequence appropriately as columns of the identity matrix are found, and selecting each pivot element so that an artificial index leaves the basic sequence whenever possible. Let $\hat{B}$ denote the final basic sequence obtained in this way. Stop if $\hat{B}$ contains no artificial indices.

Step 3. Calculate $\boldsymbol{A}_{\hat{B}}^{-1}$.

Step 4. Continue from step 1 after updating the input with the basic sequence $\hat{B}$, the basis inverse $\boldsymbol{A}_{\hat{B}}^{-1}$, and the set of previously considered indices $\hat{\mathcal{C}}$.

Though their details differ somewhat, Algorithms 3.1 and 3.5 share the same essential features. Indeed, it is easy to see that the dominant computational work in each step is the same for the two algorithms. Algorithm 3.5, however, may require n iterations (as opposed to $n-m$ for Algorithm 3.1) when applied to an $m \times n$ problem. Relying on these observations and the arguments in Proposition 3.2, we immediately have the following complexity bound:

Proposition 3.6. *Algorithm 3.5 can be used to find a basic sequence for an $m \times n$ matrix in $O\left(\left(mr + \frac{m^2T(r)}{r^3}\right)n\right)$ arithmetic operations.*

The combination of Algorithms 3.1 and 3.5 gives a procedure for solving the general basis crashing problem. Our main result on the complexity of this problem follows.

Theorem 3.7. *If we know how to multiply two $k \times k$ matrices in $O(k^{2+\delta})$ arithmetic operations, then we can solve the $m \times n$ basis crashing problem in $O(m^{\frac{3-\delta}{2-\delta}}n)$ arithmetic operations.*

Proof: Taking note of Propositions 3.2 and 3.6, we see that by combining Algorithms 3.1 and 3.5, we can construct an algorithm that solves the basis crashing problem in $O\left(\left(mr + \frac{m^2T(r)}{r^3}\right)n\right)$ arithmetic operations. Since r is a parameter of this algorithm, we are free to choose its value in a manner which minimizes the complexity bound. Clearly we should choose r so that the terms mr and $\frac{m^2T(r)}{r^3}$ are as equal as possible. To see how this can be done, it helpful to view the complexity in an alternative manner.

For any ϵ such that satisfies $T(k) = O(k^{2+\epsilon})$, the algorithm runs in $O(mr + m^2r^{\epsilon-1})$ arithmetic operations. Setting $r = \lceil m^{\frac{1}{2-\delta}} \rceil$ then gives $O(m^{\frac{3+\epsilon-2\delta}{2-\delta}} + m^{\frac{3-\delta}{2-\delta}})$ time. But since δ is necessarily an upper bound on ϵ, this last expression is $O(m^{\frac{3-\delta}{2-\delta}}n)$, as claimed. ∎

Since it is known that two $k \times k$ matrices can be multiplied in $O(k^{2.38})$ arithmetic operations [2], we have the following as a trivial corollary:

Corollary 3.8. *The $m \times n$ basis crashing problem can be solved in $O(m^{1.62}n)$ arithmetic operations.*

Finally, we note that it is has been conjectured that for every $\epsilon > 0$ there exists an algorithm that multiplies two $k \times k$ matrices in $O(k^{2+\epsilon})$ arithmetic operations. If this conjecture holds, our complexity results for basis crashing approach $O(m^{1.5}n)$ arithmetic operations.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *Journal of Symbolic Computation* **9** (1990) 251–280.

[3] G. Golub and C. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1989.

[4] W. Marlow, *Mathematics for Operations Research*, John Wiley & Sons, New York, 1978.

[5] N. Megiddo, "On finding primal- and dual-optimal bases," *ORSA Journal of Computing* **3** (1991) 63–65.

[6] V. Pan, *How to multiply natrices faster?*, Lecture Notes in Computer Science Vol. 179, Springer-Verlag, Berlin, 1984.

[7] P. Vaidya, "Speeding-up linear programming using fast matrix multiplication," in: *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Angeles, 1989, pp. 332–337.

[8] S. Winograd, *Arithmetic Complexity of Computations*, SIAM, Philadelphia, 1980.

[9] S. Winograd, "Algebraic complexity theory," in: *Handbook of Theoretical Computer Science, Volume A*, J. van Leeuwen, ed., MIT Press, Cambridge, MA, 1990, pp. 633–672.