Advanced Algorithms	**Lecture 14 : May 16, 1995**
Lecturer: Johan Håstad	*Scribe: Anders Holst*

Matrix multiplication, continued

14.1 The Complexity of Matrix Multiplication

The number of multiplications required for $n \times n$ matrices is denoted by $M(n,n,n)$ and it grows like n^w for some w. We are interested in the size of the exponent w. We will find and describe an algorithm that gives an upper bound on the size of w. The exponent w corresponding to a certain algorithm will be called the *efficiency* of the algorithm. We need also to define the efficiency of algorithms doing many disjoint matrix multiplications and also of algorithms doing only approximate matrix multiplication. The key to the current algorithm is to prove that we can convert such algorithms to exact algorithms for one matrix multiplication without decreasing the efficiency.

What we try to find is thus:

$$w = \inf \frac{\log M(n,n,n)}{\log n}.$$

The number of multiplications needed to multiply an m by p matrix by a p by r matrix is denoted by $M(m,p,r)$ and we claim that

$$M(m,p,r) \propto (mpr)^{w/3}$$

which gives

$$w = \inf 3 \frac{\log M(m,p,r)}{\log(mpr)}.$$

To see that this defines the same number note that this second definition can only give a smaller number since we have extended the range of the infimum. On the other hand know we from before that $M(mpr,mpr,mpr) \le M(m,p,r)^3$ and this gives the reverse inequality.

Now it is time for the main idea of this lecture. It is to somehow try to make more than one matrix multiplication at the same time. We need to define also the efficiency of such an algorithm. If we compute t disjoint matrices, where the i'th is of size (m_i, p_i, r_i), with in total Q multiplications, then the efficiency of that algorithm is defined to be the number w such that

$$\sum_{i=1}^{t} (m_i p_i l_i)^{w/3} = Q.$$

In our case we want to do two multiplications, one consisting of matrices with elements x_{ij} and y_{jk} which should give the elements z_{ik} as answer, and similarly one with u_{jk} and v_{ki} giving w_{ji} as answer. The indices are in the intervals $i \in [1, m]$, $j \in [1, p]$ and $k \in [1, r]$. Then we can do these calculations:

$$\sum_j (x_{ij} + u_{jk})(y_{jk} + \varepsilon v_{ki}) - \sum_j u_{jk} y_{jk} - \varepsilon v_{ki} \sum_j (x_{ij} + u_{jk}) = \sum_j x_{ij} y_{jk} = z_{ik}$$
$$\sum_k (x_{ij} + u_{jk})(y_{jk} + \varepsilon v_{ki}) - \sum_k u_{jk} y_{jk} - x_{ij} \sum_k (y_{jk} + \varepsilon v_{ki}) = \varepsilon \sum_k u_{jk} v_{ki} = \varepsilon w_{ji}$$

The first equation is done for all i and k, and the second for all i and j. They represent one $T(m, p, r)$ and one $T(p, r, m)$. Since the first two terms consist of sums of the same products in both equations, these products have only to be evaluated once. This gives a total number of multiplications $= mpr + pr + mr + pm$. Although this is already less than $2mpr$, we can do even better. Remove the third term in the first equation. This will give only an approximative answer, but it is done in $mpr + pr + pm$ multiplications.

There are two minor problems with this approach. First, we have now only an approximative algorithm for multiplication, and second, we can do two multiplications but have in the original problem only one. We will in the rest of the lecture show that we can eliminate both problems without decreasing the efficiency except by an arbitrarily small amount. For simplicity we will treat the two problems separately from each other, although it is possible to do them at the same time.

14.2 Exact multiplication with the approximative algorithm

We have an approximative algorithm that requires Q multiplications to do an T(m,p,r). (Let us forget for a while that it actually does two matrix multiplications.) We will convert this to an exact algorithm.

Note that the approximative algorithm gives us a polynomial in ε, $P(\varepsilon)$. In our case above it has degree $d = 1$. By running the algorithm twice, with *e. g.* $\varepsilon = 1$ and $\varepsilon = 2$, we can then extrapolate the two answers to get the required $P(0)$. In general, if we have an approximative algorithm that gives us a result that is a polynomial $P(\varepsilon)$ of degree d, we can run it for $d+1$ different values of ε and then extrapolate to get the exact answer. This extrapolation is only a linear combination of the individual results, $P(0) = \sum_{i=1}^{d+1} c_i P(\varepsilon_i)$, with fixed constants c_i (as soon as we have fixed all ε_i), so it requires no extra multiplications.

This means that if we have an approximative algorithm of degree d that requires Q multiplications, we can get an exact answer in $(d+1)Q$ multiplications.

So far it does not seem like a big advantage with this whole approach. We can gain about half of the multiplications if we accept an approximative answer, but then we have to run this twice instead to find out the real answer. However, the point is that we can now use the approximative algorithm recursively, and thus blow up the win in multiplications during these steps, more than we have to pay back in repeated applications of the algorithm at the top level.

Theorem 1 *If we have an approximative algorithm with efficiency w, then, for any $\delta > 0$ there exist an exact algorithm with efficiency $w + \delta$.*

Proof: Suppose we have an approximative algorithm with degree d for $T(m, p, r)$ which requires Q multiplications. Then we can take the s-fold product of the algorithm on itself, *i. e.* we run it recursively s levels. This yields an algorithm (still approximative) for $T(m^s, p^s, r^s)$ in Q^s multiplications, with degree ds. This means that we have an exact algorithm for $T(m^s, p^s, r^s)$ in $(sd+1)Q^s$ multiplications, which gives the efficiency

$$w = 3\frac{\log((sd+1)Q^s)}{\log(m^s p^s r^s)} = 3\frac{\log(sd+1) + s\log(Q)}{s\log(mpr)} \leq 3\frac{\log Q}{\log(mpr)} + \delta$$

for sufficiently large s. ■

Thus by saving half (almost) of the multiplications in the basic algorithm, we will gain in 2^s times fewer multiplications, when this is recursively applied for s levels. The cost to thereafter retrieve an exact answer will only be $s+1$. However, the gain in the basic step, is built upon doing two multiplications at the same time. In the same way, the gain of 2^s after s recursive levels, actually amounts to doing 2^s different matrix multiplications at once. Thus to gain anything at all, we must make sure that we can utilize these extra matrix multiplications.

14.3 Combining a number of disjoint multiplications into one

Suppose we can calculate two disjoint matrix multiplications $T(m_1, p_1, r_1)$ and $T(m_2, p_2, r_2)$ together with Q multiplications. This operation can be thought of as a tensor $T((m_1, p_1, r_1) \oplus T(m_2, p_2, r_2))$. If we run this recursively s levels, we get $T((m_1, p_1, r_1) \oplus T(m_2, p_2, r_2))^{\otimes s}$, which means that we have in Q^s multiplications achieved 2^s matrix multiplications. However, these are not of the same size. In more detail, $\binom{s}{i}$ of them has size $T(m_1^i m_2^{s-i}, p_1^i p_2^{s-i}, r_1^i r_2^{s-i})$.

Thus there are two steps involved here. First we have to deal with that the multiplications have different size, and thereafter we must combine several equal-size multiplications into one large matrix multiplication.

The first problem is unexpectedly simple. Of the 2^s different sized multiplications we only use those of one size. Rather than using the size that is most abundant, we will use the size that is computationally most advantageous, *i. e.* the one that maximizes

$$\binom{s}{i}(m_1p_1r_1)^{iw/3}(m_2p_2r_2)^{(s-i)w/3}.$$

Since

$$((m_1p_1r_1)^{w/3} + (m_2p_2r_2)^{w/3})^s = \sum_{i=0}^{s} \binom{s}{i}(m_1p_1r_1)^{iw/3}(m_2p_2r_2)^{(s-i)w/3}$$

we can always find an i such that

$$\binom{s}{i}(m_1p_1r_1)^{iw/3}(m_2p_2r_2)^{(s-i)w/3} \geq \frac{1}{s+1}((m_1p_1r_1)^{w/3} + (m_2p_2r_2)^{w/3})^s$$

and this will be the chosen size. It is easy to see that this implies that for any given δ, for sufficiently large s the efficiency of the algorithm has dropped by at most δ. For the special case of two disjoint matrix multiplications we have proven the following theorem:

Theorem 2 *If we have an exact algorithm for disjoint matrix multiplications with efficiency w, then, for any $\delta > 0$ there exist an exact algorithm with efficiency $w + \delta$ for disjoint matrix multiplication of one single size.*

Proof: We need only show how to modify the proof to deal with an arbitrary sum rather than a sum of two terms. Suppose we have an algorithm computing t disjoint matrix multiplications and using Q multiplications. If we take an s power of this algorithm then we obtain t^s different matrix multiplications by Q^s multiplications. The number of different sizes that occur is, however, at most s^t and hence there is one particular size that corresponds to a fraction at least s^{-t} of the "efficiency sum". Finally since

$$\frac{\log(s^t)}{s}$$

tends to 0 when s tends to infinity for sufficiently large s we have lost at most δ in the efficiency. ■

Now we can do K disjoint equal-sized matrix multiplications $T(m,p,r)$ with Q multiplications. Note that the efficiency w is the efficiency per computed matrix, $Q/K = (mpr)^{w/3}$. It is time to see what efficiency we can achieve for one single matrix multiplication.

Theorem 3 *If we have an exact algorithm for disjoint matrix multiplications of the same size with efficiency w, then, for any $\delta > 0$ there exist an exact algorithm with efficiency $w + \delta$ that computes a single matrix multiplication.*

Proof: We have an algorithm which with Q multiplications can perform K disjoint matrix multiplications $T(m,p,r)$. If we use it recursively one level, *i. e.* we try to compute $T(m^2,p^2,r^2)$, we can now manage K of the Q recursive multiplications with only one call to the algorithm. This means that it suffices with $Q/K = L$ calls (for simplicity assuming L to be an integer) and since the top level invocation of the algorithm also produces K copies, we have produced K copies of $T(m^2,p^2,r^2)$ in LQ multiplications. In general, with s recursive levels we get K copies of $T(m^s,p^s,r^s)$ in $QL^{s-1} = KL^s$ multiplications. At the top level we can just throw away $K-1$ copies and save only one. The efficiency will then be

$$w = 3\frac{\log(KL^s)}{\log(m^s p^s r^s)} = 3\frac{\log K + s\log L}{s\log(mpr)} \leq 3\frac{\log L}{\log(mpr)} + \delta$$

for sufficiently large s. (Note that the original algorithm here uses $Q/K = L$ multiplications per produced matrix, so it should say L rather than Q when comparing the efficiency of the original algorithm with the new w.)

If Q/K is not an integer we have to proceed as follows. Let $H = \lceil (Q/K) \rceil$ be the smallest integer larger than Q/K. We can then repeat the above argument with H

(simply by assuming that we did more multiplications than needed), however we have lost a little bit (since $L \leq H < L+1$) in efficiency. However, we can first take a t power of the original algorithm producing K^t copies of $T(m^t, p^t, r^t)$ to a cost of Q^t. If Let $H = \lceil (Q^t/K^t) \rceil$ and we repeat the above construction we get an efficiency

$$w = 3\frac{\log(K^t H^s)}{\log(m^{st} p^{st} r^{st})} = 3\frac{t \log K + s \log H}{st \log(mpr)} \leq 3\frac{\log L}{\log(mpr)} + \delta$$

if we first chose t sufficiently large to make H a sufficiently close approximation of L^t and then s sufficiently large to decrease the influence of K^t. ■

14.4 Conclusions

The main idea we started from, was to make two (approximative) matrix multiplications, corresponding to $T(m,p,r)$ and $T(p,r,m)$, in a total of $mpr + pr + pm$ multiplications. This gives an efficiency of

$$w = 3\frac{\log((mpr + pr + pm)/2)}{\log(mpr)}$$

One choice of m, p and r that gives an especially low value on the efficiency, is $m = 7$, $p = 1$ and $r = 7$:

$$w = 3\frac{\log((49 + 7 + 7)/2)}{\log(49)} = 3\frac{\log 31.5}{\log 49} \approx 2.659$$

According to the steps described in this lecture, this value of the efficiency propagates all the way up, to yield an exact algorithm for a single matrix multiplication, with efficiency arbitrarily close to 2.659 .

As already noted, the currently fastest known algorithm has an efficiency of 2.376. It is not known how the actual rank of the multiplication tensor $T(m,p,r)$ grows with its size, and thus not what the theoretically best efficiency is.