1 Lecture 6, November 25, 1999

The main topic of this lecture is fast matrix multiplication. This topic is covered very well in textbooks, so the notes will be more sketchy than usual, and are meant mainly to record the topics covered.

1.1 Multiplying complex numbers

Given two complex numbers, $a + \imath b$ and $c + \imath d$, we wish to compute their product $(ac - bd) + \imath(ad + bc)$. Hence we need to compute two values, $M_1 = (ac - bd)$ and $M_2 = (ad + bc)$. We assume that multiplication of real numbers is much more expensive then their addition or subtraction, and hence we wish to minimize the number of multiplications. The naive computation of M_1 and M_2 uses four multiplications. This can be reduced to three as follows. Compute $P_1 = ac$, $P_2 = bd$, and $P_3 = (a + b)(c + d)$. Now $M_1 = P_1 - P_2$ and $M_2 = P_3 - P_1 - P_2$. We have used three multiplications, two additions (for computing P_3), and three subtractions.

1.2 A more general scenario

More generally, we may study a scenario in which we are given two types of input variables, $x_1, x_2, \ldots x_c$ and $y_1, y_2, \ldots y_r$. (In the complex product case we have $x_1 = a$, $x_2 = b$, $y_1 = c$, $y_2 = d$.) We are given a list $M_1, M_2, \ldots$ of desired expressions that we need to compute, where each expression is a linear combination of cross products between x and y terms. (In the complex product case $M_1 = x_1y_1 - x_2y_2$ and $M_2 = x_1y_2 + x_2y_1$.) We wish to find a smallest l and basic products $P_1, \ldots P_l$ such that each M_i can be expressed as a linear combination of the P_j's. A basic product is a linear combination of x variables multiplied by a linear combination of y variables.

There is a nice matrix representation for this question, where the M_i are represented as c by r matrices, with entry kl in the matrix equal to the coefficient of x_ky_l in M_i. The P_j's can be chosen as arbitrary c by r *rank one* matrices. (I am too lazy to draw this here now, but illustrations appear in the CLR book.) This helps in visualizing the meaning of the various products, and can guide us in obtaining a good choice of basic products $P_1, \ldots P_l$.

1.3 Matrix multiplication

Let $X = \{x_{ij}\}$ and $Y = \{y_{ij}\}$ be two order n matrices. Their product $M = \{m_{ij}\}$ is defined via $m_{ij} = \sum_k a_{ik}b_{kj}$. Note that matrix multiplication is not a commutative operation. For computing matrix multiplication $XY = M$, Strassen used the following approach. Assume for simplicity that n is a power of 2. Break each n by n matrix into 4 $n/2$ by $n/2$ blocks. Then for $1 \leq i, j \leq 2$ indexing a block, $M_{ij} = X_{i1}Y_{1j} + X_{i2}Y_{2j}$. Hence to compute M we need to compute four expressions M_1, M_2, M_3, M_4. Each expression is a linear combination of two products. Hence altogether eight products of $n/2$ by $n/2$ matrices are involved. Strassen showed that there is a choice of seven basic products from which all expressions can be derived as linear combinations.

This improves over the naive $O(n^3)$ time for matrix multiplication as follows. Let $T(n)$

be the time to multiply two order n matrices. Than we can obtain the recursion

$$T(n) = 7T(n/2) + O(n^2)$$

where the $O(n^2)$ term acounts for the addition operations. As there are $\log n$ levels of recursion, the number of multiplications is $7^{\log n} = n^{\log 7} \simeq n^{2.81}$. The number of additions is the same, up to constant factors.

There are many possible choices of seven basic products for this problem. In class we used the visual approach to derive one such choice. Think of each quantity below as the matrix representing it. Observe that in this representation, $\sum M_{ij}$ is a matrix of rank two. Hence it is the sum of the two rank one matrices below:

$P_1 = (X_{11} + X_{21})(Y_{11} + Y_{12})$
$P_2 = (X_{12} + X_{22})(Y_{21} + Y_{22})$

It now suffices to compute only M_{11}, M_{12} and M_{21}, because M_{22} can then be obtained as

$$M_{22} = P_1 + P_2 - M_{11} - M_{12} - M_{21}$$

Hence we need to compute only three expressions, and we still have five basic products left to introduce. But we can also reuse P_1 for this purpose. Hence we will have six basic products generating three expressions that include six products altogether, which seems like a reasonable task.

There are several ways of using two basic products to generate M_{21}, and similarly for M_{12}. In anticipation of the basic product P_7 that we shall later use in order to generate M_{11}, we use the basic products

$P_3 = X_{21}(Y_{11} + Y_{21})$
$P_4 = (-X_{21} + X_{22})Y_{21}$

to generate

$$M_{21} = P_3 + P_4$$

and the basic products

$P_5 = X_{12}(Y_{12} + Y_{22})$
$P_6 = (X_{11} - X_{12})Y_{12}$

to generate

$$M_{12} = P_5 + P_6$$

To generate the missing M_{11}, it suffices to introduce just one more basic product:

$P_7 = (X_{12} + X_{21})(Y_{12} - Y_{21})$

giving

$$M_{11} = P_1 - P_3 - P_6 - P_7$$

Finally, substituting for M_{11}, M_{21}, M_{12} in the expression above for M_{22} we obtain

$$M_{22} = P_2 - P_4 - P_5 + P_7$$

Altogether we use 10 additions/subractions to generate the P_j's, and then 8 additions/subtractions to generate the M_{ij}'s.

The asymptotic running time of fast matrix multiplication has been improved by introducing more sophisticated techniques. The current best bound, by Coppersmith and Winograd, is roughly $O(n^{2.376})$.

1.4 Boolean matrix multiplication

Strassen's algorithm uses not only additions and multiplications so as to multiply matrices, but also subtractions (the inverse of addition). For Boolean matrix multiplication, the matrices have only 0/1 entries, scalar products are replaced by logical *and*, and scalar additions are replaced by logical *or*. (Hence $m_{ij} = \bigvee_k x_{ik} \wedge y_{kj}$.) Now there is no notion of subtractions, and Strassen's algorithm does not apply.

However, we can simulate Boolean matrix multiplication by integer matrix multiplication, if in the final answer we interpret every nonzero entry as 1. Hence Strassen's algorithm applies also in this case. To avoid building up large numbers in intermediate steps of the algorithm, we can perform all operations modulo k, where $k > n$ is an arbitrary integer. This does not affect the final result of the integer matrix multiplications, because all entries of the final result are smaller than k.

Another method for Boolean matrix multiplications uses only bit operations, but uses also randomization. It replaces logical *and* and *or* by multiplication and addition modulo 2. A random matrix Y' is created by chnaging every 1 entry in Y independently with probability 1/2 to 0. Then the modulu 2 product $M' = XY'$ is computed. Observe that $m'_{ij} = 1$ implies $m_{ij} = 1$, and furthermore, that $m_{ij} = 1$ implies that with probability 1/2 over the choice of Y', $m'_{ij} = 1$. Hence by repeating the experiment $O(\log n)$ times with independently chosen Y', the correct M is obtained with high probability as the logical *or* of all the M' matrices that were obtained.

1.5 Applications to graph reachability problems

Let A be an adjacency matrix of a directed graph $G(V, E)$. That is, $a_{ij} = 1$ if $(i, j) \in E$, and 0 otherwise. Then entry ij of A^k counts the number of walks from i to j (where a walk is a path that can repeat edges and vertices).

To see if a directed graph contains triangles (directed cycles of length 3), we can use fast matrix multiplication to compute A^3, and check if the diagonal has a nonzero entry. (Alternatively, compare entries ij in A^2 to entries ji in A.) Note that this is faster than exhaustively checking all triples of vertices.

If we are just interested in connectivity information, then we use Boolean matrix multiplication. If we want connectivity of distance up to k, we can add self loops to all vertices of G (adding 1 along the diagonal of A), and then A^k automatically includes all 1 entries from all A^i, $i \leq k$. To compute *transitive closure* (which pairs of vertices are connected by directed paths), compute A^n. This uses $\log n$ matrix multiplications, by repeated squaring. The naive algorithm for computing transitive closure performs breadth first search from each possible starting vertex. This takes $O(mn)$ time. The fast matrix multiplication approach is asymptotically more efficient when the graphs are dense ($m \gg n^{\omega-1}$).

References:

- A. Aho, J. Hopcroft, J. Ullman. "The Design and Analysis of Computer Algorithms". Addison-Wesley, 1974.

- T. Cormen, C. Leiserson, R. Rivest. "Introduction to Algorithms". The MIT Press and McGraw-Hill Book Company, 1990.