Lectures 1 and 2 Matrix multiplication and matrix inversion
Scribe: Jessica Su **Date:** September 25, 2015
Editor: Kathy Cooper

This class is about matrix multiplication and how it can be applied to graph algorithms.

1 Prior work on matrix multiplication

Definition 1.1. *(Matrix multiplication) Let A and B be n-by-n matrices (where entries have $O(\log n)$ bits). Then the product C, where $AB = C$ is an n-by-n matrix defined by $([i,j]) = \sum_{k=1}^{n} A(i,k)B(k,j)$.*

We will assume that operations (addition and multiplication) on $O(\log n)$ integers takes $O(1)$ time, i.e. we'll be working in a word-RAM model of computation with word size $O(\log n)$.

There has been much effort to improve the runtime of matrix multiplication. The trivial algorithm multiplies $n \times n$ matrices in $O(n^3)$ time. Strassen ('69) surprised everyone by giving an $O(n^{2.81})$ time algorithm. This began a long line of improvements until in 1986, Coppersmith and Winograd achieved $O(n^{2.376})$. After 24 years of no progress, in 2010 Andrew Stothers, a graduate student in Edinburgh, improved the running time to $O(n^{2.374})$. In 2011, Virginia Williams got $O(n^{2.3729})$, which was the best bound until Le Gall got $O(n^{2.37287})$ in 2014. Many believe that the ultimate bound will be $n^{2+O(1)}$, but this has yet to be proven.

Today we'll discuss the relationship between the problems of matrix inversion and matrix multiplication.

2 Matrix multiplication is equivalent to matrix inversion

Matrix inversion is important because it is used to solve linear systems of equations. Multiplication is equivalent to inversion, in the sense that any multiplication algorithm can be used to obtain an inversion algorithm with similar runtime, and vice versa.

2.1 Multiplication can be reduced to inversion

Theorem 2.1. *If one can invert n-by-n matrices in $T(n)$ time, then one can multiply n-by-n matrices in $O(T(3n))$ time.*

Proof. Let A and B be matrices. Let

$$D = \begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}$$

where I is the n-by-n identity matrix. One can verify by direct calculation that

$$D^{-1} = \begin{bmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix}$$

Inverting D takes $O(T(3n))$ time and we can find AB by inverting C. Note that C is always invertible since its determinant is 1. □

2.2 Inversion can be reduced to multiplication

Theorem 2.2. *Let $T(n)$ be such that $T(2n) \geq (2+\varepsilon)T(n)$ for some $\varepsilon > 0$ and all n. If one can multiply n-by-n matrices in $T(n)$ time, then one can invert n-by-n matrices in $O(T(n))$ time.*

Proof idea: First, we give an algorithm to invert symmetric positive definite matrices. Then we use this to invert arbitrary invertible matrices.

The rest of Section 2 is dedicated to this proof.

2.2.1 Symmetric positive definite matrices

Definition 2.1. *A matrix A is symmetric positive definite if*

1. *A is symmetric, i.e. $A = A^t$, so $A(i,j) = A(j,i)$ for all i, j*
2. *A is positive definite, i.e. for all $x \neq 0$, $x^t A x > 0$.*

2.2.2 Properties of symmetric positive definite matrices

Claim 1. *All symmetric positive definite matrices are invertible.*

Proof. Suppose that A is not invertible. Then there exists a nonzero vector x such that $Ax = 0$. But then $x^t A x = 0$ and A is not symmetric positive definite. So we conclude that all symmetric positive definite matrices are invertible. □

Claim 2. *Any principal submatrix of a symmetric positive definite matrix is symmetric positive definite. (An m-by-m matrix M is a principal submatrix of an n-by-n matrix A if M is obtained from A by removing its last $n-m$ rows and columns.)*

Proof. Let x be a vector with m entries. We need to show that $x^t M x > 0$. Consider y, which is x padded with $n-m$ trailing zeros. Since A is symmetric positive definite, $y^t A y > 0$. But $y^t A y = x^t M x$, since all but the first m entries are zero. □

Claim 3. *For any invertible matrix A, $A^t A$ is symmetric positive definite.*

Proof. Let x be a nonzero vector. Consider $x^t(A^t A)x = (Ax)^t(Ax) = ||Ax||^2 \geq 0$. We now show $||Ax||^2 > 0$. For any $x \neq 0$, Ax is nonzero, since A is invertible. Thus, $||Ax||^2 > 0$ for any $x \neq 0$. So $A^t A$ is positive definite. Furthermore, it's symmetric since $(A^t A)^t = A^t A$. □

Claim 4. *Let n be even and let A be an $n \times n$ symmetric positive definite matrix. Divide A into four square blocks (each one $n/2$ by $n/2$):*

$$A = \begin{bmatrix} M & B^t \\ B & C \end{bmatrix}.$$

Then the Schur complement, $S = C - BM^{-1}B^t$, is symmetric positive definite.

The proof of the above claim will be in the homework.

2.2.3 Reduction for symmetric positive definite matrices

Let A be symmetric positive definite, and divide it into the blocks M, B^t, B, and C. Again, let $S = C - BM^{-1}B^t$. By direct computation, we can verify that

$$A^{-1} = \begin{bmatrix} M^{-1} + M^{-1}B^t S^{-1} B M^{-1} & -M^{-1}B^t S^{-1} \\ -S^{-1}BM^{-1} & S^{-1} \end{bmatrix}$$

Therefore, we can compute A^{-1} recursively, as follows: (let the runtime be $t(n)$)

Algorithm 1: Inverting a symmetric positive definite matrix

Compute M^{-1} recursively (this takes $t(n/2)$ time)
Compute $S = C - BM^{-1}B^t$ using matrix multiplication (this takes $O(T(n))$ time)
Compute S^{-1} recursively (this takes $t(n/2)$ time)
Compute all entries of A^{-1} (this takes $O(T(n))$ time)

The total runtime of the procedure is

$$t(n) \leq 2t(n/2) + O(T(n)) \leq O(\sum_j 2^j T(n/2^j))$$

$$\leq O(\sum_j (2/(2+\varepsilon))^j T(n)) \leq O(T(n)).$$

2.2.4 Reduction for any matrix

Suppose that inverting a symmetric positive definite matrix reduces to matrix multiplication. Then consider the problem of inverting an arbitrary invertible matrix A. By Claim 3, we know that A^tA is symmetric positive definite, so we can easily find $C = (A^tA)^{-1}$. Then $CA^t = A^{-1}A^{-t}A^t = A^{-1}$, so we can compute A^{-1} by multiplying C with A^t.

3 Boolean Matrix Multiplication (Introduction)

Scribe: Robbie Ostrow
Editor: Kathy Cooper

Given two $n \times n$ matrices A, B over $\{0, 1\}$, we define Boolean Matrix Multiplication (BMM) as the following:

$$(AB)[i, j] = \bigvee_k (A(i, k) \wedge B(k, j))$$

Note that BMM can be computed using an algorithm for integer matrix multiplication, and so we have BMM for $n \times n$ matrices is in $O(n^{\omega+O(1)})$ time, where $\omega < 2.373$ (the current bound for integer matrix multiplication).

Most theoretically fast matrix multiplication algorithms are impractical. Therefore, so called "combinatorial algorithms" are desirable. "Combinatorial algorithm" is loosely defined, but one has the following properties:

- Doesn't use subtraction
- All operations are relatively practical (like a lookup tables)

Remark 1. *No $O(n^{3-\epsilon})$ time combinatorial algorithms for matrix multiplication are known for $\varepsilon > 0$, even for BMM! Such an algorithm would be known as "truly subcubic."*

4 Four Russians

In 1970, Arlazarov, Dinic, Kronrod, and Faradzev (who seem not to have all been Russian) developed a combinatorial algorithm for BMM in $O(\frac{n^3}{\log n})$, called the Four-Russians algorithm. With a small change to the algorithm, its runtime can be made $O(\frac{n^3}{\log^2 n})$. In 2009, Bansal and Williams obtained an improved

algorithm running in $O(\frac{n^3}{\log^{2.25} n})$ time. In 2014, Chan obtained an algorithm running in $O(\frac{n^3}{\log^3 n})$ and then, most recently, in 2015 Yu, a Stanford graduate student, achieved an algorithm that runs in $O(\frac{n^3}{\log^4 n})$. Today we'll present the Four-Russians algorithm.

4.1 Four-Russians Algorithm

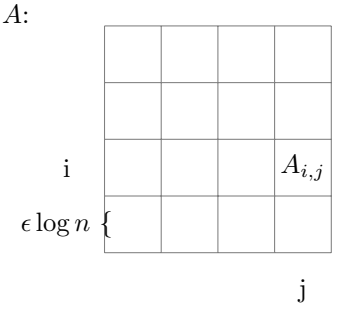
We start with an assumption:

- We can store a polynomial number of lookup tables T of size n^c where $c \leq 2 + \epsilon$, such that given the index of a table T, and any $O(\log n)$ bit vector x, we can look up $T(x)$ in constant ($O(1)$) time.

Theorem 4.1. *BMM for $n \times n$ matrices is in $O(\frac{n^3}{\log^2 n})$ time (on a word RAM with word size $O(\log n)$).*

Proof. We give the Four Russians' algorithm. (More of a description of the algorithm than a full proof of correctness.)

Let A and B be $n \times n$ boolean matrices. Choosing an arbitrary ϵ, we can split A into blocks of size $\epsilon \log n \times \epsilon \log n$. That is, A is partitioned into blocks $A_{i,j}$ for $i, j \in [\frac{n}{\epsilon \log n}]$. Below we give a simple example of A:



For each choice of i, j we create a lookup table $T_{i,j}$ corresponding to $A_{i,j}$ with the following specification:

For every bit vector v with length $\epsilon \log n$:

$T_{i,j}[v] = A_{i,j} \cdot v$.

That is, $T_{i,j}$ takes keys that are $\epsilon \log n$-bit sequences and stores $\epsilon \log n$-bit sequences. Also since there are n^ϵ bit vectors of $\epsilon \log n$ bits, and $A_{i,j} \cdot v$ is $\epsilon \log n$ bits, we have $|T_{i,j}| = n^\epsilon \epsilon \log n$.

The entire computation time of these tables is asymptotically

$$(\frac{n}{\log n})^2 n^\epsilon \log^2 n = n^{2+\epsilon},$$

since there are $(\frac{n}{\log n})^2$ choices for i, j, n^ε vectors v, and for each $A_{i,j}$ and each v, computing $A_{i,j}v$ take $O(\log^2 n)$ time for constant ε.

Given the tables that we created in subcubic time, we can now look up any $A_{ij} \cdot v$ in constant time.

We now consider the matrix B. Split each column of B into $\frac{n}{\epsilon \log n}$ parts of $\varepsilon \log n$ consecutive entries. Let B_j^k be the j^{th} piece of the k^{th} column of B. Each $A_{ij}B_j^k$ can be computed in constant time, because it can be accessed from $T_{ij}[B_j^k]$ in the tables created from preprocessing.

To calculate the product $Q = AB$, we can do the following.

From $j = 1$ to $\frac{n}{\epsilon \log n}$: $Q_{ik} = Q_{ik} \vee (A_{ij} \wedge B_j^k)$, by the definition. With our tables T, we can calculate the bitwise "and" (or *) in constant time, but the "or" (or sum) still takes $O(\log n)$ time. This gives us an algorithm running in time $O(n \cdot \frac{n}{\log n}^2 \cdot \log n) = O(\frac{n^3}{\log n})$ time, the original result of the four Russians.

How can we get rid of the extra $\log n$ term created by the sum?

We can precompute all possible pairwise sums! Create a table S such that $S(u, v) = u \vee v$ where $u, v \in \{0,1\}^{\epsilon \log n}$. This takes us time $O(n^{2\epsilon} \epsilon \log n)$, since there are $n^{2\epsilon}$ pairs u, v and each component takes only $O(\log n)$ time.
This precomputation allows us constant time lookup of any possible pairwise sum of $\epsilon \log n$ bit vectors.

Hence, each $Q_{ik} = Q_{ik} \vee (A_{ij} \wedge B_j^k)$ operation takes $O(1)$ time, and the final algorithm asymptotic runtime is

$$n \cdot (n/\varepsilon \log n)^2 = n^3 / \log^2 n,$$

where the first n counts the number of columns k of B and the remaining term is the number of pairs i, j.

Thus, we have a combinatorial algorithm for BMM running in $O(\frac{n^3}{\log^2 n})$ time.

□

Note: can we save more than a log factor? This is a major open problem.

5 Transitive Closure

Definition 5.1. *The Transitive Closure (TC) of a directed graph $G = (V, E)$ on n nodes is an $n \times n$ matrix such that* $\forall u, v \in E$ $(T(u, v) = \begin{cases} 1 & \text{if } v \text{ is reachable from } u \\ 0 & \text{otherwise} \end{cases}$

Transitive Closure on an undirected graph is trivial in linear time– just compute the connected components. In contrast, we will show that for directed graphs the problem is equivalent to BMM.

Theorem 5.1. *Transitive closure is equivalent to BMM*

Proof. We prove equivalence in both directions.

Claim 5. *If TC is in $T(n)$ time then BMM on $n \times n$ matrices is in $O(T(3n))$ time.*

Proof. Consider a graph like the one below, where there are three rows of n vertices. Given two boolean matrices A and B, we can create such a graph by adding an edge between the i^{th} vertex of the first row and the j^{th} of the second row iff $A_{ij} == 1$. Construct edges between the second and third rows in a similar fashion for B. Thus, the product AB can be computed by taking the transitive closure of this graph. It is equivalent to BMM by simply taking the $\wedge$ of $ij \to jk$. Since the graph has $3n$ nodes, given a $T(n)$ algorithm for TC, we have an $O(T(3n))$ algorithm for BMM. (Of course, it takes n^2 time to create the graph, but this is subsumed by $T(n)$ as $T(n) \geq n^2$ as one must at least print the output of AB.)

□

Claim 6. *If BMM is in $T(n)$ time, such that $T(n/2) \leq T(n)/(2+\epsilon)$, then TC is in $O(T(n))$ time.*

We note that the condition in the claim is quite natural. For instance it is true about $T(n) = n^c$ for any $c > 1$.

Proof. Let A be the adjacency matrix of some graph G. Then $(A + I)^n$ is the transitive closure of G. Since we have a $T(n)$ algorithm for BMM, we can compute $(A + I)^n$ using $\log n$ successive squarings of $A + I$ in $O(T(n) \log n)$ time. We need to get rid of the log term to show equivalence.
We do so using the following algorithm:

1. Compute the strongly connected components of G and collapse them to get G', which is a D.A.G. We can do this in linear time.

2. Compute the topological order of G' and reorder vertices according to it (linear time)

3. Let A be the adjacency matrix of G'. $(A+I)$ is upper triangular. Compute $C = (A+I)^*$, i.e. the TC of G'.

4. Uncollapse SCCs. (linear time)

All parts except (3) take linear time. We examine part (3).

Consider the matrix $(A+I)$ split into four sub-matrices M, C, B, and 0 each of size $n/2 \times n/2$.

$(A+I) = \begin{bmatrix} M & C \\ 0 & B \end{bmatrix}$

We claim that $(A+I)^* = \begin{bmatrix} M^* & M^*CB^* \\ 0 & B^* \end{bmatrix}$

The reasoning behind this is as follows. Let U be the first $n/2$ nodes in the topological order, and let V be the rest of the nodes. Then M is the adjacency matrix of the subgraph induced by U and B is the adjacency matrix induced by V. The only edges between U and V go from U to V. Thus, M^* and U^* represents the transitive closure restricted to $U \times U$ and $V \times V$. For the TC entries for $u \in U$ and $v \in V$, we note that the only way to get from u to v is to go from u to possibly another $u' \in U$ using a path within U, then take an edge (u', v') to a node $v' \in V$ and then to take a path from v' to v within V. I.e. the $U \times V$ entries of the TC matrix are exactly M^*CB^*.

Suppose that the runtime of our algorithm on n node graphs is $\text{TC}(n)$. To calculate the transitive closure matrix, we recursively compute M^* and B^*. Since each of these matrices have dimension $n/2$, this takes $2TC(n/2)$ time.

We then compute M^*CB^*, which takes $O(T(n))$ time, where $T(n)$ was the time to compute the boolean product of $n \times n$ matrices.

Finally, we have $TC(n) \leq 2TC(\frac{n}{2}) + O(T(n))$. If we assume that there is some $\varepsilon > 0$ such that $T(n/2) \leq T(n)/(2+\epsilon)$, then the recurrence solves to $TC(n) = O(T(n))$.

□

It follows from claim 1 and claim 2 that BMM of $n \times n$ matrices is equivalent in asymptotic runtime to TC of a graph on n nodes. □

6 Other Notes

BMM can also solve problems that look much simpler. For example: does a directed graph have a triangle? We can easily solve with an algorithm for BMM by taking the adjacency matrix, cubing it, and checking if the resulting matrix contains a 1 in the diagonal.

Somewhat surprisingly, it is also known that for any $\epsilon > 0$, an $O(n^{3-\epsilon})$ time combinatorial algorithm for triangle finding also implies an $O(n^{3-\epsilon/3})$ time combinatorial algorithm for BMM. Hence in terms of combinatorial algorithms BMM and triangle finding are "subcubic"-equivalent.