This class is about matrix multiplication and how it can be applied to graph algorithms. We will also consider faster approximation algorithms that solve problems without resorting to matrix multiplication technique.

1 Prior work on matrix multiplication

Definition 1.1. (Matrix multiplication) Let $A$ and $B$ be $n$-by-$n$ matrices with entries over a field $K$. Then the product $C$, where $AB = C$ is an $n$-by-$n$ matrix defined by $C[i, j] = \sum_{k=1}^{n} A(i, k) \cdot B(k, j)$. Here + and $\cdot$ are operations over $K$.

If $K$ is an arbitrary field, we will assume that addition and multiplication of field elements takes $O(1)$ time. If $K$ is the field of rationals, we will assume that operations (addition and multiplication) on $O(\log n)$ bit numbers 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 follows the definition and multiplies $n \times n$ matrices in $O(n^3)$ time. Strassen (1969) 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. There are no nontrivial lower bounds except in specialized models of computation.

Today we’ll discuss the relationship between the problems of matrix inversion and matrix multiplication, and also that between Boolean matrix multiplication and triangle detection.

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 a nonsingular $n$-by-$n$ matrix in $T(n)$ time, then one can multiply $n$-by-$n$ matrices in $O(T(3n))$ time.

Proof. Let $A$ and $B$ be matrices. Consider the following $3n \times 3n$ matrix:

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

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 this section 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 $x \neq 0$, $x^tAx > 0$.

#### 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 $x^tAx = 0$ and $A$ is not symmetric positive definite. So we conclude that all symmetric positive definite matrices are invertible. \hfill \Box

**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 $x^tMx > 0$. Consider $y$, which is $x$ padded with $n - m$ trailing zeros. Since $A$ is symmetric positive definite, $y^tAy > 0$. But $y^tAy = x^tMx$, since all but the first $m$ entries are zero. \hfill \Box

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

**Proof.** Let $x$ be a nonzero vector. Consider $x^t(A^tA)x = (Ax)^t(Ax) = ||Ax||^2 \geq 0$. 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 $A^tA$ is positive definite. Furthermore, it’s symmetric since $(A^tA)^t = A^tA$. \hfill \Box

**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

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

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

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

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

Note that BMM can be computed using an algorithm for integer matrix multiplication, and so we have that BMM for $n \times n$ matrices is in $n^{\omega + o(1)}$ 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 $O(n^{3-\varepsilon})$ time combinatorial algorithms for matrix multiplication are known for $\varepsilon > 0$, even for BMM! Such an algorithm would be known as “truly subcubic.”

Next lecture we will see some slightly subcubic combinatorial algorithms for BMM. Today we will consider the relationship between triangle detection and BMM.
The triangle detection problem is as follows: Given an undirected graph \( G = (V, E) \), determine whether \( G \) contains \( a, b, c \in V \) so that \( (a, b), (b, c), (c, a) \in E \).

There is a simple reduction from triangle detection to BMM: given an \( n \) node \( G \), define its Boolean adjacency matrix \( A \) as the \( n \times n \) matrix with \( A[i, j] = 1 \) if \( (i, j) \in E \) and \( A[i, j] = 0 \) otherwise. Consider \( A^3 = A \cdot A \cdot A \).

\[
A^3[i, i] = \bigvee_{j, k} A[i, j] \wedge A[j, k] \wedge A[k, i],
\]

which is 1 if \( i \) is contained in a triangle in \( G \) and 0 otherwise. Thus, we can find a node \( i \) that lies in a triangle (if one exists) by multiplying \( A \) by itself 3 times and looking at the diagonal. If such an \( i \) is found, one can clearly also find a triangle that contains \( i \) with an additional \( O(n^2) \) search. Thus even finding a triangle in \( G \) can be computed in the time that it takes to multiply two \( n \times n \) matrices over the Boolean semiring.

An important open question is whether triangle detection is equivalent to BMM. We will show an equivalence in the following sense: if there is a combinatorial algorithm for triangle detection running in truly subcubic time, then there is also a combinatorial BMM algorithm running in truly subcubic time.

**Theorem 3.1.** Suppose that a triangle in an \( n \) node graph can be detected in \( O(n^{3-\varepsilon}) \) time for some \( \varepsilon > 0 \). Then BMM of two \( n \times n \) matrices can be computed in \( O(n^{3-\varepsilon/3}) \) time.

Before we prove the theorem, let us cast the BMM problem as a graph problem. Suppose that we are given two \( n \times n \) Boolean matrices \( A \) and \( B \). Define the graph \( H \) as a tripartite graph on partitions \( I, J, K \) of size \( n \) each. \( I \) corresponds to the rows of \( A \), \( K \) corresponds to the columns of \( B \) and \( J \) corresponds to the columns of \( A \) and rows of \( B \).

There are no edges within \( I, J, K \) and the edge \((i, k)\) is in \( H \) for every \( i \in I, k \in K \). For \( i \in I, j \in J \), \((i, j) \in H \) if and only if \( A[i, j] = 1 \), and for \( j \in J, k \in K \), \((j, k) \in H \) if and only if \( B[j, k] = 1 \).

Now, an edge \((i, k)\) is in a triangle in \( H \) (for \( i \in I, k \in K \)) if and only if \((AB)[i, k] = 1 \). In other words, the BMM problem is exact that of computing all pairs \((i, k) \in I \times K \) such that \((i, k)\) is in a triangle in \( H \).

Given this, we have a very simple algorithm that computes BMM using an oracle for triangle detection:

“While \( H \) contains a triangle, find a triangle \((i, j, k)\), set \((AB)[i, k] = 1 \), remove \((i, k)\) from \( H \)”

The algorithm is correct. However, it is not very efficient. It runs up to \( n^2 \) queries of triangle detection in an \( n \) node graph, so in general it would run in \( \Omega(n^3) \) time. Now, notice that any reduction from BMM to triangle detection must make \( n^2 \) queries in the worst case since BMM needs to return \( n^2 \) bits, while a call to triangle detection only reveals 1 bit. However, none said that we need to run the query on a large graph. We will thus improve the above algorithm by reducing the size of the graphs that we query.

Below we will assume that if the triangle detection algorithm says that the graph has a triangle, we can also find such a triangle in the same time. We encourage you to think why this is true - i.e. why one can reduce triangle finding to detection so that if one can detect in \( t(n) \) time, then one can also find in \( O(t(n)) \) time.

Split \( I, J \) and \( K \) into \( t \) pieces each on \( n/t \) nodes. Call the pieces of \( I \), \( \{I_i\}_i \), those of \( J \), \( \{J_j\}_j \), and those of \( K \), \( \{K_k\}_k \). Now run the following algorithm:

(0) \( C \) - all zeroes \( n \times n \) matrix (will be the output)
(1) Go through all \( t^3 \) triples of pieces \((I_i, J_j, K_k)\):
  (1.1) While the subgraph of \( H \) induced by \( I_i \cup J_j \cup K_k \) contains a triangle:
    (1.1.1) find this triangle \((a, b, c) \in I_i \times J_j \times K_k\)
    (1.1.2) set \( C[a, c] = 1 \)
    (1.1.3) remove \((a, c)\) from the global graph \( H \).

The algorithm is clearly correct - for any \((a, c)\), if it is in a triangle with some \( b \), then when the triple \((I_i, J_j, K_k)\) such that \( a \in I_i, b \in J_j, c \in K_k \) is considered, either a triangle for \((a, c)\) has already been found,
or \((a, b, c)\) is in the graph induced by \(I_i \cup J_j \cup K_k\) so that a triangle including \((a, c)\) will be found in this iteration.

What is the runtime?

Every time a triangle detection call finds a triangle, it sets an entry of \(C\) to 1, and since the corresponding edge is removed globally, each entry of \(C\) is set to 1 at most once. Thus, if finding a triangle in an \(N\) node graph takes \(T(N)\) time, then the runtime due to YES instances of triangle detection is \(n^2T(n/t)\). Some triangle detection calls might return NO. However this happens at most once per triple \((I_i, J_j, K_k)\). Thus the runtime due to NO instances of triangle detection is \(t^3T(n/t)\). We set \(t^3 = n^2\) to minimize the runtime, and we get that BMM can be computed in \(O(n^2T(n^{1/3}))\) time.