1 Boolean Matrix Multiplication (Introduction)

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 BMM for $n \times n$ matrices is in $O(n^\omega)$ time, where $\omega < 2.373$ (the current bound for integer matrix multiplication).

Most theoretical fast matrix multiplication algorithms are impractical. Therefore, so called “combinatorial algorithms” are desirable. The term “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.”

2 BMM is subcubically equivalent to graph triangle detection.

Last time we set up the proof for the following theorem:

**Theorem 2.1.** 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.

Today we will finish the proof.

Suppose that we are given an algorithm that can detect a triangle in $O(n^{3–\varepsilon})$ time for some $\varepsilon > 0$. We will actually 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.

Now recall the graph representation of the BMM problem: given two $n \times n$ matrices $A$ and $B$, build a graph $H$ on node partitions $I, J, K$ where we put an edge from $i \in I$ to $j \in J$ iff $A[i, j] = 1$ and similarly from $j \in J$ to $k \in K$ iff $B[j, k] = 1$. Finally, we put an edge $(i, k)$ for every pair $i \in I, k \in K$, and the BMM problem becomes, for every edge $(i, k)$ with $i \in I, k \in K$, is $(i, k)$ contained in some triangle $i, j, k$?

Now, let $t$ be a parameter we will set later. Split $I, J$ and $K$ into $t$ pieces each on $n/t$ nodes. Let $I_i, J_i,$ and $K_i$ be the $i^{th}$ piece of $I, J,$ and $K$ respectively. Now run the following algorithm:

1. (0) $C$ - all zeroes $n \times n$ matrix (will be the output)
2. (1) Go through all $t^3$ triples of pieces $(I_i, J_j, K_k)$:
   1. (1.1) While the subgraph of $H$ induced by $I_i \cup J_j \cup K_k$ contains a triangle:
      1. (1.1.1) find this triangle $(a, b, c) \in I_i \times J_j \times K_k$
      1. (1.1.2) set $C[a, c] = 1$
      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 \(O(N^3)\) time for \(\varepsilon > 0\), then the runtime due to YES instances of triangle detection is \(O(n^2(n^{1/3})^{3-\varepsilon})\). 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 \(O(t^3(n/t)^{3-\varepsilon})\). We set \(t^3 = n^2\) to minimize the runtime, and we get that BMM can be computed in \(O(n^2(n^{1/3})^{3-\varepsilon})\) time.

### 3 The Four Russians Algorithm

In 1970, Arlazarov, Dinic, Kronrod, and Faradzev (who seem not to have all been Russian) developed a combinatorial algorithm for BMM in \(O(n^3 \log n)\), called the Four-Russians algorithm. With a small change to the algorithm, its runtime can be made \(O(n^3 \log 2 n)\). In 2009, Bansal and Williams obtained an improved algorithm running in \(O(n^3 \log^2 n)\) time. In 2014, Chan obtained an algorithm running in \(O(n^3 \log^2 n)\) and then, most recently, in 2015 Huacheng Yu achieved an algorithm that runs in \(O(n^3 \log^3 n)\). Today we’ll present the Four-Russians algorithm.

#### 3.1 Four-Russians Algorithm

We start with an assumption:

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

This assumption is true in the word RAM model with \(O(\log n)\) bit words.

**Theorem 3.1.** Under the assumption, BMM for \(n \times n\) matrices is in \(O(n^3 \log^2 n)\) time.

**Proof.** We give the Four Russians’ algorithm.

Let \(A\) and \(B\) be \(n \times n\) boolean matrices. Choosing an arbitrary \(\epsilon\), 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 \(i, j \in \lfloor n/\epsilon \log n \rfloor\). Below we give a simple example of \(A\):

\[
\begin{array}{ccc}
\text{i} \\
\epsilon \log n \\
\hline
\end{array}
\begin{array}{cccc}
\text{A}_{i,j} \\
\text{j} \\
\end{array}
\]

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 \(v\) of 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 \(n^\epsilon\) 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
\[ \left( \frac{n}{\log n} \right)^2 n^\epsilon \log^2 n = n^{2+\epsilon}, \]
since there are \( \frac{n}{\log n} \) choices for \( i, j \), \( n^\epsilon \) vectors \( v \), and for each \( A_{i,j} \) and each \( v \), computing \( A_{i,j} \cdot v \) takes \( O(\log^2 n) \) time for constant \( \epsilon \).

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 \( \epsilon \log n \) consecutive entries. Let \( B^k_j \) be the \( j^{th} \) piece of the \( k^{th} \) column of \( B \). Each \( A_{ij} \cdot B^k_j \) can be computed in constant time, because it can be accessed from \( T_{ij}[B^k_j] \) in the tables created from preprocessing.

To calculate the product \( Q = AB \), we can do the following.

From \( j = 1 \) to \( \frac{n}{\epsilon \log n} \) : \( Q_{ik} = Q_{ik} \lor (A_{ij} \cdot B^k_j) \), by the definition (here \( \cdot \) is Boolean matrix-vector multiplication). With our tables \( T \), we can calculate each \( A_{ij} \cdot B^k_j \) in constant time, but the bitwise “or” of \( Q_{ik} \) and \( A_{ij} \cdot B^k_j \) still takes \( O(\log n) \) time. This gives us an algorithm running in time \( O(n \cdot \frac{n}{\epsilon \log n} \cdot \log n) = O(\frac{n^3}{\log n}) \) 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 bitwise ORs! Create a table \( S \) such that \( S(u, v) = u \lor v \) 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 \( Q_{ik} = Q_{ik} \lor (A_{ij} \land B^k_j) \) operation takes \( O(1) \) time, and the final algorithm asymptotic runtime is
\[ n \cdot (n/\epsilon \log n)^2 = n^3 / \log^2 n, \]
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 \( O(\frac{n^3}{\log n}) \) time.

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

### 3.2 Yu’s algorithm

The best known combinatorial algorithm for BMM is by Yu in 2015. In the following theorem statement, \( \tilde{O} \) notation hides factors polynomial in \( \log \log n \).

**Theorem 3.2.** BMM for \( n \times n \) matrices is in \( \tilde{O}(n^3/(\log n)^4) \) time.

We will not prove this theorem but the high-level idea of the proof is the following. The proof is stated in terms of the triangle detection problem. It uses a high-degree low-degree technique, which is a common technique in graph algorithms. First, we look for a high-degree vertex. If we find one, we use a divide conquer technique. On the other hand, if all of the vertices have low-degree, we use a lookup table technique.
4 BMM and the Triangle Removal Lemma

The triangle removal lemma is an important lemma in combinatorics and it turns out to be related to BMM algorithms. In particular, Bansal and Williams’s BMM algorithm from 2009 is based on this connection.

Roughly speaking, the triangle removal lemma says that if a graph doesn’t have too many triangles then you can remove a small number of edges to get a triangle-free graph. More formally, it says the following.

Lemma 4.1 (Triangle removal lemma (Ruzsa, Szemerédi)). For all $\epsilon > 0$, there exists $\delta(\epsilon)$ such that if an $n$-node graph $G = (V, E)$ has at most $\epsilon n^3$ triangles then there exists $E' \subseteq E$ with $|E'| \leq \delta n^2$ such that $G \setminus E'$ has no triangles.

Note that $\delta$ is a function of $\epsilon$. The best known dependence of $\delta$ on $\epsilon$ is $1/\delta \sim 2^{O((\log^*(1/\epsilon)))}$ by Jacob Fox. The definition of $\log^* x$ is the number of times one must apply $\log$ to $x$ in order to obtain a constant. Gowers showed that one would not be able to get $1/\delta$ to be bigger than $2^{\delta(\sqrt{\log n})}$. It is possible, however, that $1/\delta$ can be made to be $2^{\Theta(\sqrt{\log n})}$ and this would be a substantial improvement in the Triangle removal lemma.

It would also have nice algorithmic consequences for triangle detection and BMM, as shown by Bansal and Williams [2009].

We will show that if there were to be a better triangle removal lemma, the “wish-triangle removal lemma”, then there would be a faster combinatorial BMM algorithm.

Wish-Triangle removal lemma. There exists a $O(\delta n^3)$ time algorithms such that for all $\epsilon > 0$, there exists $\delta(\epsilon) < (1/\log^4 n)$ such that if an $n$-node graph $G = (V, E)$ has at most $\epsilon n^3$ triangles then the algorithm produces a set $E'$ of at most $\delta n^2$ edges such that $G \setminus E'$ has no triangles.

Claim 1. If the wish-triangle removal lemma is true, then there’s a combinatorial algorithm for BMM running in $\tilde{O}(\delta \cdot n^3)$ time that is correct with high probability ($\geq 1 - 1/poly(n)$).

Proof. The algorithm is as follows. For a constant $c$, sample $c\sqrt{n} \ln n$ triples of vertices $(i, j, k)$ and check whether any of them form a triangle.

Case 1. $G$ has $\geq n^{2.5}$ triangles. We will show that it is likely that we sampled one. The probability we didn’t sample any of them is $(1 - n^{2.5}/n^3)^{c\sqrt{n} \ln n} = ((1 - 1/\sqrt{n})^{c \ln n} \leq (1/e)^{c \ln n} = n^{-c}$.

Case 2. $G$ has $< n^{2.5}$ triangles. In this case, we use the wish-triangle removal lemma algorithm with $\epsilon = 1/\sqrt{n}$ to get $\delta(1/\sqrt{n}) \cdot n^2$ edges $E'$ such that $G \setminus E'$ has no triangles. This takes time $O(\delta(1/\sqrt{n}) \cdot n^3)$. Then, for all $e \in E'$, $v \in V$, check if $e, v$ forms a triangle. This also takes time $O(\delta(1/\sqrt{n}) \cdot n^3)$.

Note that the choice of $\sqrt{n}$ above was arbitrary. We could have chosen any $n^c$ for $c < 3$, and we would get a subcubic algorithm from the Wish-triangle removal lemma.