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:

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

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 $O(n^\omega)$ 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.

We will prove the following theorem.

Theorem 2.1. *Suppose that a triangle in an n node graph can be detected in $D(n)$ time. Then BMM of two $n \times n$ matrices can be computed in $O(n^2 D(n^{1/3}))$ time.*

In particular, if $D(n) = O(n^{3-\varepsilon})$ time, then BMM is in $O(n^{3-\varepsilon/3})$ time, and if $D(n) = O(n^3/\log^c n)$ time for some constant c, then BMM is in $O(n^3/\log^c n)$ time, asymptotically the same as $D(n)$.

Suppose that we are given an algorithm that can detect a triangle in $D(n)$ time. 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 consider the following 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:

(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_j) 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 $D(N)$ time, then the runtime due to YES instances of triangle detection is $O(n^2 D(n/t))$. 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 $O(t^3 D(n/t))$.

We set $t^3 = n^2$ to minimize the runtime, and we get that BMM can be computed in $O(n^2 D(n^{1/3}))$ 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 running in $O(\frac{n^3}{\log n})$ time, now 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 Huacheng Yu achieved an algorithm that runs in $O(\frac{n^3}{\log^4 n})$. Today we'll present the Four-Russians algorithm and briefly discuss Yu's algorithm.

3.1 Four-Russians Algorithm

We start with an assumption:

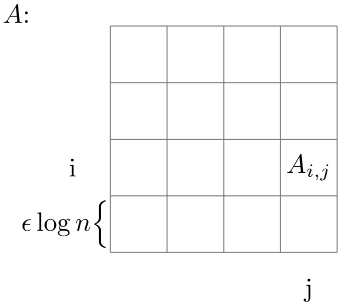
- We can store a polynomial number of lookup tables (arrays) T of size n^c where $c \leq 2 + \epsilon$, such that given an index of a table T, and any $O(\log n)$ bit vector x, we can look up $T(x)$ in constant ($O(1)$) time.

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 $O(\frac{n^3}{\log^2 n})$ time.*

Proof. We give the Four Russians' algorithm.

Let A and B be $n \times n$ boolean matrices. Choosing an arbitrary $\epsilon > 0$, 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 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^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

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

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 $A_{i,j} \cdot v$ takes $O(\log^2 n)$ 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 $\frac{n}{\epsilon \log n}$: $Q_{ik} = Q_{ik} \vee (A_{ij} \cdot B_j^k)$, by the definition (here $\cdot$ is Boolean matrix-vector multiplication). With our tables T, we can calculate each $A_{ij} \cdot B_j^k$ in constant time, but the bitwise "or" of Q_{ik} and $A_{ij} \cdot B_j^k$ 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 bitwise ORs. 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 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, $\hat{O}$ notation hides factors polynomial in $\log \log n$.

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

Yu actually obtains an $\hat{O}(n^3/(\log n)^4)$ time algorithm that can detect triangles in an n-node graphs. By the equivalence we proved earlier, this immediately implies a BMM algorithm of the same complexity.

Suppose we are given a tripartite graph G with vertex partitions A, B, C and we want to know if there is a triangle in G; since the graph is tripartite the triangle will have nodes $a \in A, b \in B, c \in C$.

Yu's algorithm considers the vertices of A. It has two cases. Either every single vertex in A has somewhat low degree, or there is some vertex in A that has somewhat high degree. The first case uses a Four-Russians' style algorithm. The second case uses recursion.

Let's look at the low degree part first. For a subset of vertices X and a node v, let $deg_X(v)$ denote the number of neighbors of v in X.

Lemma 3.1. *Let $G = (V, E)$ be a tripartite graph with partitions A, B, C with sizes k, m, n respectively. Suppose that for some D, for all $a \in A$, $d_B(a) \cdot d_C(a) \leq mn/D^2$. Then there is an $O(mnD^{6D} + kmn/D^4 + k(m+n))$ time combinatorial algorithm that can detect a triangle on a word-RAM with words of size $\Omega(D \log D + \log(kmn))$.*

If $N = \max\{k, m, n\}$ and D is small enough, say $O(\log N/(\log\log N))$, then the word size of the word-RAM only needs to be $\Theta(\log N)$. We will apply the lemma exactly for such parameters.

Let's prove the lemma. Partition B into groups B_i of size D^3. Do the same with C, obtaining n/D^3 such groups C_j.

Define S_B to be the set of sets S of size at most D s.t. S is entirely contained in some B_i. There are $O((m/D^3)\binom{D^3}{D})$ such sets S, and so this is the size of S_B.

Similarly, define S_C to be the set of sets S of size at most D s.t. S is entirely contained in some C_j. Then $|S_C| \leq O((n/D^3)\binom{D^3}{D})$.

For every $S \in S_B$ and $S' \in S_C$, we check if there is an edge in $S \times S'$ and store the results in a look-up table. The computation takes $O\left(D^2 \cdot (mn/D^6) \cdot \binom{D^3}{D}^2\right) \leq O(mnD^{6D})$ time, and this is also the size of the look-up table.

We can index any subset $S \in S_B \cup S_C$ using $O(D\log(D) + \log(m+n))$ bits, which is the size of the word in the word-RAM.

To check if there is a triangle using this look-up table, we do the following. For every $a \in A$, partition the neighbors $N_B(a)$ of a in B into at most $m/D^3 + deg_B(a)/D$ sets of size at most D, each in S_B. We do this by partitioning, for each B_i, the neighbors $N_B(a) \cap B_i$ greedily into sets of size D and at most one set of size $< D$.

Similarly, we split the neighbors $N_C(a)$ of a in C into at most $n/D^3 + deg_C(a)/D$ sets in S_B (of size at most D). Then for each pair of such sets, we look up in the look-up table whether there is an edge between them. The total time over all a is asymptotically

$$\sum_{a \in A}((m/D^3 + deg_B(a)/D)(n/D^3 + deg_C(a)/D) + k(m+n)) \leq$$

$$kmn/D^6 + 2kmn/D^4 + k(m+n) + \sum_{a \in A} deg_B(a) deg_C(a)/D^2 \leq$$

$$O(kmn/D^4 + k(m+n)).$$

The total running time together with the preprocessing becomes $O(mnD^{6D} + kmn/D^4 + k(m+n))$.

This finishes the low-degree phase. We set $D = \log(N)/(10\log\log N)^2$. Then the running time becomes $mnN^{o(1)} + \hat{O}(kmn/\log^4(N) + k(m+n))$.

What does Yu's algorithm do if there is a node a in A that has high degree, i.e. $deg_B(a) deg_C(a) > mn/D^2$? Then it picks this node, lets B_1 be the neighbors of a in B, and C_1 be the neighbors of a in C.

If B_1 is a larger fraction of B than C_1 is of C, i.e. if $|B_1|/|B| > |C_1|/|C|$, then we recurse on $(A \setminus \{a\}, B, C \setminus C_1)$ and $(A \setminus \{a\}, B \setminus B_1, C_1)$. (Otherwise, reverse the roles of B and C and recurse accordingly.)

If either recursive step returns YES, return YES. Otherwise, if both return NO, check all pairs of vertices in $B_1 \times C_1$ to see if there is an edge completing a triangle with a. If so, return YES, otherwise return NO.

The idea is that we will do the work for $B_1 \times C_1$ at the end if needed. Otherwise, we can avoid it. If we have checked $A \setminus \{a\}$ for triangles with $B \times (C \setminus C_1)$, then we only need to check it for triangles with $(B \setminus B_1) \times C_1$ since $B_1 \times C_1$ can be avoided.

(The base case is when $|B|$ or $|C|$ becomes smaller than D^6 when one can use the brute force algorithm.)

The running time analysis is the most complicated part. We don't have time to cover it, but by careful analysis of the recursion tree, Yu is able to show that the running time is $\hat{O}(n^3/\log^4 n)$.