1 Triangle Detection vs Matrix Multiplication

Some problems can be solved via matrix multiplication, resulting in $O(n^3)$ time complexity. The question is whether one could maybe get a faster than $O(n^3)$-time algorithm without matrix multiplication, or if we should just stay with the matrix-based algorithm and use existing libraries that optimize matrix multiplication (numpy, BLAS etc).

Definition 1.1 (Boolean Matrix Multiplication)**.** *Given two $n \times n$ matrices $A, B \in \{0,1\}^{n \times n}$, we define the boolean matrix product $C \in \{0,1\}^{n \times n}$ as*

$$C_{i,j} = \begin{cases} 1 & \text{if } (AB)_{i,j} \neq 0 \\ 0 & \text{else} \end{cases}$$

For example, $C_{i,j} = \min(1, (AB)_{i,j})$. We only care about the zero/non-zero entries, but not the exact value of the non-zero entries.

For intuition, boolean matrix multiplication can be interpreted as multiplying two matrices with non-negative entries, and then asking which entries of the result are non-zero. I.e. we just care about the zero vs. non-zero profile of the matrix but not the actual numbers involved. Example:

$$\begin{bmatrix} 0 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 10 & 0 \end{bmatrix} \qquad \text{regular product}$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \qquad \text{boolean product}$$

Theorem 1.2 ([WW10])**.** *If there exists an algorithm that can detect a triangle in an n-node graph in $O(n^{3-\epsilon})$ time, then there exists an algorithm for computing a boolean matrix product in $O(n^{3-\epsilon/3} \log n)$ time.*

The reduction is somewhat surprising given that triangle detection gives only a single bit of information "is there a triangle, or not". Yet somehow we can blow up that single bit into n^2 many bits, since the result of a matrix product consists of n^2 entries.

Note that the opposite direction is easy: For a graph $G = (V, E)$ we can find a triangle by computing just 2 matrix products. Given adjacency matrix $\mathbf{A}$, we compute $\mathbf{A}^2$ and check for each $\{u, v\}$ if $(\mathbf{A}^2)_{u,v} \neq 0$. So if we can multiply matrices in less than $O(n^3)$ time, then we can find a triangle in less than $O(n^3)$ time.

The above theorem says that the reverse is also true: If we can detect a triangle in subqubic time, then we can multiply matrices in subqubic time.

On an intuitive level, the theorem says that if we manage to develop some new fast algorithm for triangle detection, then we indirectly also invented a novel way to multiply matrices. However, matrix multiplication is fundamental within computer science and developing fast algorithms for matrix multiplication is a very active fields of research. So beating existing work will definitely be difficult. For us, it would be easier to just use their existing fast matrix multiplication results and use them to detect triangles.

To prove the above theorem, we start with the following lemma.

Lemma 1.3. *Given a triangle detection algorithm with complexity $O(T(n))$, two boolean matrices $\mathbf{A}, \mathbf{B}$, and a set $M \subset \{1, ..., n\} \times \{1, ..., n\}$. We can find all $(i, j) \in M$ with $(\mathbf{A} \cdot \mathbf{B})_{i,j} \neq 0$ in time $O(T(n) + (T(n) \cdot k \cdot \log n)$, where k is the number of such non-zero entries.*

Proof. We construct a graph with vertices $u_1, ..., u_n, v_1, ..., v_n, w_1, ..., w_n$. We add edges (u_i, v_j) for all i, j with $\mathbf{A}_{i,j} = 1$, and edges (v_i, w_j) for all i, j with $\mathbf{B}_{i,j} = 1$, and edges (w_i, u_j) for all $(i, j) \in M$.

There exists a triangle in this graph, if and only if there is some $(i, j) \in M$ with $(\mathbf{A} \cdot \mathbf{B})_{i,j} = 1$. This is because $(\mathbf{A} \cdot \mathbf{B})_{i,j} = \sum_k \mathbf{A}_{i,k} \mathbf{B}_{k,j}$ which is non-zero if and only if there is k with $\mathbf{A}_{i,k} = \mathbf{B}_{k,j} = 1$, i.e. we have a triangle $u_i v_k w_j$.

We can now find all the indices $(i, j) \in M$ with $(\mathbf{A} \cdot \mathbf{B})_{i,j} = 1$ via binary search on set M. □

Proof of Theorem 1.2. Given $\mathbf{A}, \mathbf{B}$, we split the matrices into blocks of size $n^{1/3} \times n^{1/3}$. Let's call these blocks $\mathbf{A}^{i,j}, \mathbf{B}^{i,j}$ for $i, j = 1, ..., n^{2/3}$. Let $\mathbf{C} = \mathbf{AB}$ and perform the same split on $\mathbf{C}$, so we have

$$\mathbf{C}^{i,j} = \sum_{k=1}^{n^{2/3}} \mathbf{A}^{i,k}\mathbf{B}^{k,j}.$$

Observe that since we only care about the non-zero entries we do not need to compute $(\mathbf{A}^{i,k}\mathbf{B}^{k,j})_{a,b}$ for $a, b \in \{1, ..., n^{1/3}\}$ if we know $(\mathbf{A}^{i,k'}\mathbf{B}^{k',j})_{a,b} \neq 0$ for some $k' \neq k$. That's because if $(\mathbf{A}^{i,k'}\mathbf{B}^{k',j})_{a,b} \neq 0$, then we already know that $\mathbf{C}^{i,j}_{a,b} \neq 0$, so we do not need to compute $(\mathbf{A}^{i,k}\mathbf{B}^{k,j})_{a,b}$ anymore.

Thus we can compute all $\mathbf{C}^{i,j}$ via the following algorithm.

- For $i, j = 1, ..., n^{2/3}$:
  - //We now compute $C^{i,j}$
  - $M = \{1, ..., n^{1/3}\} \times \{1, ..., n^{1/3}\}$
  - //We now check for all $(a, b) \in M$ whether $C^{i,j}_{a,b} \neq 0$.
  - For $k = 1, ..., n^{2/3}$
    - Find all non-zeros $(\mathbf{A}^{i,k}\mathbf{B}^{k,j})_{a,b}$ for $(a, b) \in M$ via Lemma 1.3.
    - For the discovered non-zero entries a, b, set $C^{i,j}_{a,b} = 1$ and remove (a, b) from M.

Let $K_{i,j,k}$ be the number of non-zero we found in product $\mathbf{A}^{i,k}\mathbf{B}^{k,j}$. Since we only look for non-zeros in set M (which is initially of size $(n^{1/3})^2$) and remove each (a, b) if $(\mathbf{A}^{i,k}\mathbf{B}^{k,j})_{a,b} \neq 0$, we have for all i, j that

$$\sum_{k=1}^{n^{2/3}} K_{i,j,k} = (n^{1/3})^2 = n^{2/3}.$$

Hence, by Lemma 1.3 the total time complexity of this is

$$O((n^{2/3})^2(\sum_k K_{i,j,k} T(n^{1/3}) \log n)) = O(n^{4/3}(n^{2/3}n^{(3-\epsilon)/3}) \log n) = O(n^{3-\epsilon/3} \log n).$$

□

References

[WW10] Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *FOCS*, pages 645–654. IEEE Computer Society, 2010.