An Improved Combinatorial Algorithm for Boolean Matrix Multiplication

Huacheng Yu*
Stanford University

February 26, 2017

Abstract

We present a new combinatorial algorithm for triangle finding and Boolean matrix multiplication that runs in $\hat{O}(n^3/\log^4 n)$ time, where the $\hat{O}$ notation suppresses poly(loglog) factors. This improves the previous best combinatorial algorithm by Chan [4] that runs in $\hat{O}(n^3/\log^3 n)$ time. Our algorithm generalizes the divide-and-conquer strategy of Chan's algorithm.

Moreover, we propose a general framework for detecting triangles in graphs and computing Boolean matrix multiplication. Roughly speaking, if we can find the "easy parts" of a given instance efficiently, we can solve the whole problem faster than n^3.

1 Introduction

Boolean matrix multiplication (BMM) is one of the most fundamental problems in computer science. It has applications to triangle finding, transitive closure, context-free grammar parsing, etc [5, 7, 10, 11]. One way to multiply two Boolean matrices is to treat them as integer matrices, and apply a fast matrix multiplication algorithm over the integers. Matrix multiplication can be done in "truly subcubic time", i.e., the product of two $n \times n$ matrices can be computed in $O(n^{3-\epsilon})$ additions and multiplications over the field. For example, the latest generation of such algorithms run in $O(n^{2.373})$ operations [9, 12]. These algorithms are "algebraic", as they rely on the structure of the field, and in general the ring structure of matrices over the field.

There is a different group of BMM algorithms, often called "combinatorial" algorithms. They usually reduce the redundancy in the computation by exploiting some combinatorial structure in the Boolean matrices.[1] The "Four Russians" algorithm by Arlazarov, Dinic, Kronrod, and Faradzhev [1] is the most well-known combinatorial algorithm for BMM. On the RAM model with word size $w = \Theta(\log n)$, the "Four Russians" algorithm can be implemented in $O(n^3/\log^2 n)$ time. About 40 years later, this result was improved by Bansal and Williams [2]. They presented an $O(n^3(\log\log n)^2/\log^{9/4} n)$ time combinatorial algorithm for Boolean matrix multiplication, using the weak regularity lemma for graphs. Recently, Chan presented an $O(n^3\,(\log\log n)^3/\log^3 n)$ time algorithm [4], improving the running time even further.

Although these combinatorial algorithms have worse running times than the algebraic ones, they generally have some nice properties. Combinatorial algorithms usually can be generalized in ways that the algebraic ones cannot be. For example, Chan's algorithm partly extends an idea of divide-and-conquer in an algorithm for the offline dominance range reporting problem by Impagliazzo, Lovett, Paturi, and Schneider [8]; the algebraic structure of dominance reporting is completely different from BMM's. Moreover, in practice, these combinatorial algorithms are usually fast and easy to implement, while in contrast, most theoretically fast matrix multiplication algorithms are impractical to implement. Finding a matrix multiplication algorithm that is both "good" in theory and practice is still an important open goal of the area.

*Supported in part by NSF CCF-1212372.

[1] Unfortunately, we do not have a formal definition for "combinatorial" yet, which is an important open question.

In this paper, we generalize the ideas of Impagliazzo et al. [8] and Chan [4], to present a faster combinatorial algorithm for triangle detection: *given an n-node graph, does it contain a triangle?*

Theorem 1. *Given a tripartite graph G on n vertices, we can detect if there is a triangle in G using a combinatorial algorithm in $\hat{O}\left(n^3/\log^4 n\right)$ time on a word RAM with word size $w \geq \Omega(\log n)$.*[2]

Vassilevska Williams and Williams [13] proved that triangle detection and Boolean matrix multiplication are "subcubic equivalent" in the following sense: for any constant c, if we can solve triangle detection on n-node graphs in $O(n^3/\log^c n)$ time, we can also solve Boolean matrix multiplication on $n \times n$ matrices in the same running time. Together with Theorem 1, this gives a fast combinatorial algorithm for Boolean matrix multiplication.

Theorem 2. *There is a combinatorial algorithm to multiply two $n \times n$ Boolean matrices in $\hat{O}\left(n^3/\log^4 n\right)$ time.*

Moreover, we generalize the algorithm, and propose a general framework for solving triangle detection combinatorially.

Definition 1. *Let G be a tripartite graph on vertices $A \cup B \cup C$, G' be a subgraph of G on vertices $A' \cup B' \cup C'$, where $A' \subseteq A, B' \subseteq B, C' \subseteq C$. We say G' is* (α, β, γ)-fraction *of G if $\{\alpha, \beta, \gamma\} = \{|A'|/|A|, |B'|/|B|, |C'|/|C|\}$.*

In the other words, G' is (α, β, γ)-fraction of G if one of the vertex sets of G' has α-fraction of the corresponding set in G, another has β-fraction, and the third has γ-fraction.

Theorem 3. *Let n be an integer, $0 < c, \epsilon \leq 1$, and G be any tripartite graph on vertex sets A, B, C with at least $\sqrt{n}$ vertices in each part. If there is an algorithm $\mathbf{L}$ which takes one such G as input, and outputs a pair (G', b) such that*

- *G' is a subgraph, and is (α, β, γ)-fraction of G,*
- *$b = 1$ if G' is triangle-free, and $b = 0$ if G' contains a triangle,*
- *it runs in $O(c\alpha\beta\gamma|A||B||C|)$ time*

for some $0 < \alpha, \beta, \gamma \leq 1$ such that $\alpha\beta > 10\epsilon(1 + \log \frac{1}{\gamma})$ and $\alpha > 10\epsilon(1 + \log \frac{1}{\beta})$. Then we can solve triangle detection on n-node graphs in $O(cn^3 + n^{3-\epsilon/2})$ time.

That is, to derive an efficient algorithm for triangle finding, it is sufficient to find and solve an "easy part" of the input. This opens a new direction for attacking this problem.

Related work. Bansal and Williams [2] used the weak regularity lemma of Frieze and Kannan [6] to discover and exploit small substructures in the graph. Generally speaking, a regularity lemma partitions the vertex set of a graph into disjoint sets, so that the edge distribution between any two sets is "close to random." Bansal and Williams enumerate every triple of sets in the partition: if the subgraph induced by the triple is sparse, finding a triangle in this triple is easy. Otherwise, since the induced subgraph is dense and "close to random", the regularity lemma guarantees that it is impossible to check many pairs of vertices *without* finding an edge between them. Integrating the method of Four Russians with the above approach yields an $\hat{O}(n^3/\log^{9/4} n)$ time algorithm for triangle detection.

Chan [4] used a very different approach for triangle detection which we now outline briefly.[3] Consider a tripartite graph on vertex sets (A, B, C) such that $|A| \leq \text{polylog}(|B| + |C|)$ (if this is not the case, partition the set A into $\text{polylog}(|B| + |C|)$-size subsets, and solve them independently). If the edge set between A and B is sparse, triangle detection is easy. Otherwise, there is some node $v \in A$ with many neighbors in B. Then the algorithm checks for an edge between every pair of neighbors of v and does two recursive calls. One is on $A \setminus \{v\}$, B and the non-neighbors of v in C, and the other is on $A \setminus \{v\}$, non-neighbors of v in B and neighbors in C. On one hand, this recursive procedure never puts any pairs of neighbors of v in the branch. This guarantees that the algorithm only manually checks every pair in $B \times C$ at most once. On the other hand, the procedure may duplicate the set of non-neighbors in B when doing

[2] We use $\hat{O}(f(n))$ to suppress $\text{poly}(\log \log f(n))$ factors in the running time.

[3] To keep consistency, the following description will be in the language of triangle finding, although his paper originally presented the algorithm in the language of Boolean matrix multiplication.

a recursive call, which increases the total input size. However, since we have a lower bound on the degree of vertex v to B, the procedure does not duplicate too many vertices each time. A careful analysis shows that the overhead of the recursion is actually rather tiny.

In Section 2, we show how to extend the idea of divide-and-conquer in Chan's algorithm to get an even faster algorithm for triangle detection (and hence for Boolean matrix multiplication). We give a more intuitive proof of the recursion involving less calculation. In Section 3, we propose a general framework for solving triangle detection, as a starting point for future work in this area.

2 Triangle Detection

Preliminaries and notations. We shall use the fact that the triangle detection problem in general undirected graphs is time-equivalent to the problem restricted to tripartite graphs, up to a constant factor.[4] Henceforth, we will assume the input graph is tripartite, and the tri-partition of its vertices is given to us.

For a graph $G = (V, E)$, a vertex $v \in V$ and a subset of vertices $S \subseteq V$, we denote $d(v, S) = |E \cap (\{v\} \times S)|$, which we call the *degree of v to S*.

Main results. In what follows, we present an $\hat{O}(n^3/\log^4 n)$ time combinatorial algorithm for triangle detection.

Suppose we are given a tripartite graph G on vertex sets A, B, C. One (naive) approach to detect if there is a triangle in the graph is: for vertex $v \in A$, and all pairs (u, w) of v's neighbors, check if there is an edge between u and w. The amount of work we do for vertex v is proportional to the number of edges between v and B (the degree of v to B) times the number of edges between v and C (the degree of v to C). This approach is efficient whenever the product of these two degrees is low on average. However, if this is not the case, there must be a vertex $v \in A$ with a large product of degrees. If the enumeration reports no edge between any pair of v's neighbors, we know there has to be a large "non-edge area" between B and C, i.e. between v's neighbors. In the rest of the algorithm, there is no need to look again at any pair of vertices in that area. We implement this idea by recursion: find disjoint subsets of $B \times C$ which together cover all pairs outside the non-edge area, and recurse on them. We show this recursion is actually efficient.

Before stating and proving the efficiency of our main algorithm, we first show that the naive approach proposed above is "Four-Russianizable" as expected. That is, we can apply the Method of Four Russians to speed up the sparse case by a factor of roughly $\log^2 n$.

The following algorithm is a generalization of an algorithm of Bansal and Williams [2].

Lemma 1. *Let $G = (V, E)$ be tripartite on vertex sets A, B, C of sizes k, m, n respectively. If there is a Δ such that for all $v \in A$ we have $d(v, B)d(v, C) \leq \frac{nm}{\Delta^2}$, then we can detect if there is a triangle in G combinatorially in $O\left(mnk^{0.9} + \frac{kmn}{\log^4 k} + \frac{kmn}{\Delta^2 \log^2 k} + k(m+n)\right)$ time, on a word RAM with word size $w \geq \Omega(\log kmn)$.*

Proof. First, partition the vertices in B (resp. C) into groups $\{B_i\}$ (resp. $\{C_i\}$) of sizes $\log^3 k$ arbitrarily, e.g. put $(i \log^3 k + 1)$-th to $(i+1)\log^3 k$-th vertex of B (resp. C) in B_i (resp. C_i). Let $\mathcal{S}_B = \{S : |S| \leq \frac{\log k}{5 \log\log k}, S \subseteq B_i \text{ for some } B_i\}$, $\mathcal{S}_C = \{S : |S| \leq \frac{\log k}{5 \log\log k}, S \subseteq C_i \text{ for some } C_i\}$ be collections of subsets within the same group of B or C with at most $\frac{\log k}{5 \log\log k}$ vertices. For every $S \in \mathcal{S}_B, S' \in \mathcal{S}_C$, we determine if there is at least one edge between them, and store all the results in a lookup table. This preprocessing takes

$$O\left(\frac{mn}{\log^6 k}\left(\frac{\log^3 k}{\frac{\log k}{5 \log\log k}}\right)^2 \cdot \log^4 k\right) \leq O(mnk^{0.9})$$

time. Note that we can index a subset using $O(\log \max\{k, m, n\}) = O(w)$ bits. This table can be stored in the memory so that one table lookup takes constant time.

[4] We can always create a new graph by making three copies of the vertex set, and connecting vertices with an edge before between different parts. There is a triangle in this new tripartite graph if and only if there is one in the original graph.

With the help of this table, we can check if there is a triangle in G efficiently. We go over all vertices $v \in A$, and partition its neighborhood into a minimum number of sets in $\mathcal{S}_B, \mathcal{S}_C$. That is, for every group of vertices, we arbitrarily partition v's neighborhood in this group into sets of size exactly $\frac{\log k}{5 \log \log k}$ and (possibly) one more set of size at most $\frac{\log k}{5 \log \log k}$. This generates at most $\hat{O}\left(\frac{m}{\log^3 k} + \frac{d(v,B)}{\log k}\right)$ sets from $\mathcal{S}_B$ and at most $\hat{O}\left(\frac{n}{\log^3 k} + \frac{d(v,C)}{\log k}\right)$ sets from $\mathcal{S}_C$. Using the lookup table, we can detect if there is an edge between any pair of sets from $\mathcal{S}_B$ and $\mathcal{S}_C$ in constant time. Going over all $v \in A$ takes

$$\begin{aligned}
&\hat{O}\left(\sum_{v \in A}\left(\frac{m}{\log^3 k} + \frac{d(v,B)}{\log k}\right)\left(\frac{n}{\log^3 k} + \frac{d(v,C)}{\log k}\right) + k(m+n)\right) \\
&\leq \hat{O}\left(\frac{kmn}{\log^6 k} + \sum_{v \in A}\frac{m}{\log^3 k} \cdot \frac{n}{\log k} + \sum_{v \in A}\frac{m}{\log k} \cdot \frac{n}{\log^3 k} + \sum_{v \in A}\frac{mn}{\Delta^2 \log^2 k} + k(m+n)\right) \\
&\leq \hat{O}\left(\frac{kmn}{\log^4 k} + \frac{kmn}{\Delta^2 \log^2 k} + k(m+n)\right)
\end{aligned}$$

time. The total running time is at most $O\left(mnk^{0.9} + \frac{kmn}{\log^4 k} + \frac{kmn}{\Delta^2 \log^2 k} + k(m+n)\right)$ as we stated.

□

Using this algorithm for the sparse case as a subroutine, we give a fast combinatorial algorithm for triangle detection.

Theorem 1. *Given a tripartite graph G on n vertices, we can detect if there is a triangle in G using a combinatorial algorithm in $\hat{O}\left(n^3/\log^4 n\right)$ time on a word RAM with word size $w \geq \Omega(\log n)$.*

Proof. Set parameter $\Delta = \frac{\log n}{100(\log \log n)^2}$. It will remain fixed as we do the recursion, even if the instance size shrinks. The following algorithm detects if there is a triangle in a tripartite graph with vertex sets A, B, C:

Step 0: If $|B| < \Delta^6$ or $|C| < \Delta^6$, solve the instance by exhaustive search and return the answer.

Step 1: If for all vertices $v_i \in A$, $d(v_i, B)d(v_i, C) \leq \frac{|B| \cdot |C|}{\Delta^2}$, we solve the instance by the algorithm in Lemma 1.

Step 2: Otherwise, find a vertex that violates the condition. Without loss of generality, assume v_1 does, $d(v_1, B)d(v_1, C) > \frac{|B| \cdot |C|}{\Delta^2}$.

Step 3: Let B_1(resp. C_1) be v_1's neighborhood in B(resp. C).
If $\frac{|B_1|}{|B|} > \frac{|C_1|}{|C|}$, then recurse on $(A \setminus \{v_1\}, B, C \setminus C_1)$ and $(A \setminus \{v_1\}, B \setminus B_1, C_1)$, else recurse on $(A \setminus \{v_1\}, B \setminus B_1, C)$ and $(A \setminus \{v_1\}, B_1, C \setminus C_1)$.
Return YES if either of the two recursions returned YES.

Step 4: Check all pairs of vertices in $B_1 \times C_1$ for an edge.
Return YES if there is an edge, NO otherwise.

The correctness of the algorithm is straightforward, as it never loses track of any triples that could possible form a triangle.

Analysis of the running time. In each node of the recursion tree, only some of the following four subprocedures are executed:

1. If either $|B|$ or $|C|$ is small, we do exhaustive search, which takes $O(|A||B||C|)$ time.

2. We spend $O\left(|A|(|B|+|C|)\right)$ time to check whether there is a high degree vertex and generate the inputs for two recursive calls.

3. If A has no high degree nodes, we invoke the algorithm in Lemma 1 which takes

$$\begin{aligned}&\hat{O}\left(|B||C||A|^{0.9}+\frac{|A||B||C|}{\log^4|A|}+\frac{|A||B||C|}{\Delta^2\log^2|A|}+|A|(|B|+|C|)\right)\\&=\hat{O}\left(|B||C|n^{0.9}+n|B||C|/\log^4 n+n(|B|+|C|)\right)\end{aligned}$$

   time.

4. For every pair of neighbors of v_1 in B and C, we check if they have an edge. This step takes $O(|B_1||C_1|)$ time.

To analyse the total running time, we are going to bound the time we spend on the small-graph case (Subprocedure 1) and the time we spend on the large-graph case (Subprocedure 2,3,4) in the entire execution of the algorithm (the whole recursion tree) separately, and sum up these two cases.

For the case when $|B|\geq\Delta^6$ and $|C|\geq\Delta^6$, Subprocedure 2 is cheap compared to the other steps, taking time $O(n(|B|+|C|))\leq\hat{O}\left(n|B||C|/\log^6 n\right)$. In this case, we charge all the running time to the pairs of vertices in $B\times C$, then sum up over all pairs the cost they need to pay. If A has no high degree vertices, we will run Subprocedure 2 and 3, which takes at most $\hat{O}(|B||C|n^{0.9}+n|B||C|/\log^4 n+n(|B|+|C|))\leq\hat{O}\left(n|B||C|/\log^4 n\right)$ time. We charge this running time to all pairs of vertices in $B\times C$ evenly; that is, every pair of vertices gets charged $\hat{O}\left(n/\log^4 n\right)$. If A has a high degree vertex, we will run Subprocedure 2 and 4, which takes at most

$$\begin{aligned}&O\left(|A|(|B|+|C|)+|B_1||C_1|\right)\\&\leq\hat{O}\left(\Delta^2|B_1||C_1|n/\log^6 n+|B_1||C_1|\right)\\&\leq\hat{O}\left(n|B_1||C_1|/\log^4 n\right)\end{aligned}$$

time. We charge this running time to the pairs in $B_1\times C_1$ evenly, so every pair gets charged $\hat{O}\left(n/\log^4 n\right)$.

Claim 1. *Every pair of vertices is charged at most once, over the entire execution of the algorithm.*

The proof of this claim follows from inspection of the algorithm: the above argument only charges pairs of vertices that are not going into the same recursive branch.

There are n^2 pairs at the very beginning. Every pair gets charged at most $\hat{O}\left(n/\log^4 n\right)$. Therefore the running time for the large-graph case is at most $\hat{O}\left(n^3/\log^4 n\right)$.

Next we bound the total running time of Subprocedure 1. This running time is proportional to the number of triples we enumerated in Step 0. Let $T(S)$ be the maximum possible of this number of triples, if we start our recursion from vertex sets A,B,C with $|B||C|\leq S$.

Claim 2. $T(S)\leq nS$. *Moreover, if* $S>n\Delta^6$, *then*

$$T(S)\leq\max_{t>1/\Delta,t'>1/\Delta^2}\left\{T((1-t')S)+T\left(t'(1-t)S\right)\right\}$$

Proof. $T(S)\leq nS$ is trivial, since we never enumerate any triple more than once. If $S>n\Delta^6$, we have $|B|,|C|>\Delta^6$. That is, we must be starting the recursion from a large-graph case. There is nothing to prove if there is no high degree vertex in A, as we will do no enumeration in Step 0. Otherwise, let $t=\max\left\{\frac{|B_1|}{|B|},\frac{|C_1|}{|C|}\right\},t'=\min\left\{\frac{|B_1|}{|B|},\frac{|C_1|}{|C|}\right\}$, then by the algorithm, we have $tt'>1/\Delta^2$, in particular, $t>1/\Delta,t'>1/\Delta^2$. In the two recursive calls, we have $|B||C|$ values $(1-t')S$ and $t'(1-t)S$ respectively. By the definition of T, we will enumerate at most $T\left((1-t')S\right)+T\left(t'(1-t)S\right)$ triples. This proves the claim.

□

We can upper bound $T(S)$ using this recurrence. Consider the recursion tree $\mathcal{R}$ for $T(S)$. The root has value S. Its left child has value $(1-t')S$ and right child has value $t'(1-t)S$, where t and t' maximize $T((1-t')S) + T(t'(1-t)S)$. We recursively construct the tree for the left child and right child. For a node with value x, we always put $(1-t')x$ in its left child and $t'(1-t)x$ in its right child, for x's optimal parameter t' and t. We expand the tree from nodes with value at least $n^{1.5}(> n\Delta^6)$ recursively. Therefore, we will get a tree with leaf values at most $n^{1.5}$. This tree demonstrates how $T(S)$ is computed according to the recurrence, before we reach $n^{1.5}$. By the construction of the tree, the T-value of a node is upper-bounded by the sum of T-values of its two children. Since for leaves, we have $T(x) \le nx$, the sum of values of all leaves multiplied by n is an upper bound for $T(S)$.

We calculate this sum by distinguishing two cases. For every leaf, there is a unique path from the root to it, in which we follow the left child in some steps, follow the right child in the rest. Consider all leaves such that the unique path from the root to it takes at most $10\Delta\log\Delta(\le \frac{\log n}{10\log\log n})$ right-child-moves. Since $t' > 1/\Delta^2$, the depth of the tree is at most $\Delta^2 \ln\sqrt{n} < \Delta^2 \log n$. Therefore, there are at most

$$\begin{aligned}\binom{\Delta^2\log n}{10\Delta\log\Delta}\cdot 10\Delta\log\Delta &\le \left(\frac{e\Delta^2\log n}{10\Delta\log\Delta}\right)^{10\Delta\log\Delta}\cdot 10\Delta\log\Delta\\ &\le \left(\log^2 n\right)^{10\Delta\log\Delta}\cdot 10\Delta\log\Delta\\ &\le 2^{\frac{1}{5}\log n}\cdot o(\log n) \le O(n^{0.4})\end{aligned}$$

such leaves. By construction, each leaf has value at most $n^{1.5}$, so the sum over all leaves is at most $O(n^{1.9})$.

For those leaves such that the path from the root takes more than $10\Delta\log\Delta$ right-child-moves, consider a tree $\mathcal{R}'$ with *identical structure* as $\mathcal{R}$, but we set the values of nodes in a different way. We set all ts to 0 but leave all t's unchanged, then calculate the corresponding values. That is, the root still has value S, its left child still has value $(1-t')S$, and its right child now has value $t'S$. For any node with value x in $\mathcal{R}'$, its left child has value $(1-t')x$ and right child has value $t'x$, for the same ratio t' as the corresponding node in $\mathcal{R}$, although the value x may have changed. Also, leaves now may have values greater than $n^{1.5}$ by the construction of $\mathcal{R}'$. The tree $\mathcal{R}'$ has the following two properties:

1. The sum of values in all leaves is exactly S.
2. For any node such that the path from root to it takes at least k right-child-moves, its value is at least $1/(1-1/\Delta)^k$ times the value of the corresponding node in $\mathcal{R}$.

The first property can be proved by induction, since the sum of values of two children is exactly the value of the parent. For the second property, since we require $t > 1/\Delta$ in $\mathcal{R}$, and keep t' unchanged, set t to 0, we will gain a factor of $1/(1-t) > 1/(1-1/\Delta)$ for every right-child-move.

By property 2 above, for all leaves that the path from root takes more than $10\Delta\log\Delta$ right-child-moves, their values in $\mathcal{R}'$ must be at least $1/(1-1/\Delta)^{10\Delta\log\Delta} \ge \Delta^{10}$ times the corresponding values in $\mathcal{R}$. However, the sum of these values in $\mathcal{R}'$ is at most S by property 1. Therefore, the sum of values in $\mathcal{R}$ is at most S/Δ^{10}. Summing up these two cases, we prove that $T(S) \le nS/\Delta^{10} + O(n^{2.9})$. In particular, $T(n^2) \le n^3/\Delta^{10} + O(n^{2.9}) = \hat{O}(n^3/\log^{10} n)$.

Finally, we sum up the small-graph and large-graph cases, proving that the algorithm runs in $\hat{O}(n^3/\log^4 n)$ time. □

Vassilevska Williams and Williams [13] showed a reduction from triangle detection to BMM.

Theorem 4 (Vassilevska Williams and Williams'10)**.** *For any constant c, if we can solve triangle detection on n-node graphs in $O(n^3/\log^c n)$ time, we can also solve Boolean matrix multiplication on $n\times n$ matrices in the same running time.*

Combining the algorithm with the reduction, we get an efficient combinatorial algorithm for BMM.

Theorem 2. *There is a combinatorial algorithm to multiply two $n\times n$ Boolean matrices in $\hat{O}\left(n^3/\log^4 n\right)$ time.*

3 A General Approach

In this section, we propose a more general approach for triangle finding, which may lead to an even faster combinatorial algorithm.

In the algorithm presented in Section 2, finding a high degree vertex $v \in A$ and its neighborhood B_1, C_1 can be viewed as finding a large easy part of the input. That is, for the subgraph induced by vertices A, B_1, C_1, there is a 2-path for every pair in $B_1 \times C_1$. Thus, we only have to spend $O(|B_1||C_1|)$ time to determine if there is a triangle in it, which is $O(1/|A|)$ time on average for every triple of vertices. After solving this part of the input, we do two recursive calls which together exactly cover the rest of the triples. The high-degree of v guarantees that the easy part we find each time cannot be too small. We will have saved enough time before reaching the case where B or C is close to constant size (in which we basically have no way to beat the exhaustive search). However, if all vertices have low degree, then that instance itself is easy (via Lemma 1). The following theorem generalizes this idea of reducing the triangle detection problem to finding a large subgraph on which triangle detection is easy.

Theorem 3. *Let n be an integer, $0 < c, \epsilon \leq 1$, and G be any tripartite graph on vertex sets A, B, C with at least $\sqrt{n}$ vertices in each part. If there is an algorithm $\mathbf{L}$ which takes one such G as input, and outputs a pair (G', b) such that*

- *G' is a subgraph, and is (α, β, γ)-fraction of G,*
- *$b = 1$ if G' is triangle-free, and $b = 0$ if G' contains a triangle,*
- *it runs in $O(c\alpha\beta\gamma|A||B||C|)$ time*

for some $0 < \alpha, \beta, \gamma \leq 1$ such that $\alpha\beta > 10\epsilon(1 + \log \frac{1}{\gamma})$ and $\alpha > 10\epsilon(1 + \log \frac{1}{\beta})$. Then we can solve triangle detection on n-node graphs in $O(cn^3 + n^{3-\epsilon/2})$ time.

Proof. First note that if $\epsilon < \frac{1}{\log n}$, the theorem is trivial, since $n^{3-\epsilon/2} = \Theta(n^3)$. In the rest of the proof, we will assume $\epsilon \geq \frac{1}{\log n}$, and thus $\alpha\beta > \frac{10}{\log n}$.

The following divide-and-conquer algorithm detects if there is a triangle among vertices $A \cup B \cup C$:

Step 0. If $|A||B||C| < n^{2.5}$, do exhaustive search on all triples and return the answer.

Step 1. Run $\mathbf{L}$ on the graph induced by $A \cup B \cup C$.
Let G' on $A' \cup B' \cup C'$ be the subgraph $\mathbf{L}$ outputs.
Return YES if G' contains a triangle.

Step 2. Rename the sets of vertices such that $|A'| = \alpha|A|, |B'| = \beta|B|, |C'| = \gamma|C|$.
Recurse on vertex sets $(A, B, C \setminus C')$, $(A, B \setminus B', C')$, $(A \setminus A', B', C')$.
Return whether any of the three recursive calls returned YES.

Its correctness is straightforward, as we never miss any triple in $A \times B \times C$. In the following, we will prove that this algorithm is efficient.

For the large-graph case, a similar "charging argument" works here as in the proof of Theorem 1. Note that the input $A \cup B \cup C$ we fed to $\mathbf{L}$ in Step 1 always has at least $\sqrt{n}$ vertices in each part. By our assumption, Step 1 will run in $O(c\alpha\beta\gamma|A||B||C|) \leq O(c|A'||B'||C'|)$ time. In Step 2, the algorithm generates the input for recursive calls, which takes only $O(|A| + |B| + |C|)$ time.[5] We charge the running time of Step 1 and Step 2 to the triples in $A' \times B' \times C'$. On average, each triple is charged $O(c)$ time. Same as the proof of Theorem 1, no triple in $A' \times B' \times C'$ will be considered again. Therefore, every triple is charged at most once in this argument. There are n^3 triples in total. They are charged at most $O(cn^3)$ time. This proves that Step 1 and Step 2 take at most $O(cn^3)$ time in the entire algorithm.

[5] It is dominated by the running time of Step 1. Since to determine if G' has a triangle, in worst case, $\mathbf{L}$ has to take at least $\Omega(|A'||B'|)$ time. But we have $|A'||B'| \geq \alpha\beta|A||B| \geq 10n^{1.5}/\log n \geq \Omega(n)$.

For the small-graph case, the time we spend on Step 0 is proportional to the number of triples we enumerated. Let $T(S)$ be the maximum possible value of this number, if we start our recursion with $|A||B||C| = S$. On the one hand, we have $T(S) \leq S$, as every triple will be manually checked at most once. Moreover, for $S \geq n^{2.5}$, by the way that the algorithm does recursive calls, we have $T(S) \leq T\left((1-\gamma)\,S\right) + T\left(\gamma(1-\beta)S\right) + T\left(\beta\gamma(1-\alpha)S\right)$.

We are going to prove $T(S) \leq n^{2.5}\left(\frac{S}{n^{2.5}}\right)^{1-\epsilon}$ by induction on S. For $S \leq n^{2.5}$, the new upper bound automatically holds, since $T(S) \leq S \leq n^{2.5}\left(\frac{S}{n^{2.5}}\right)^{1-\epsilon}$. Otherwise, by the recurrence and induction hypothesis,

$$\begin{aligned} T(S) &\leq n^{2.5}\left(\left(\frac{(1-\gamma)\,S}{n^{2.5}}\right)^{1-\epsilon} + \left(\frac{\gamma(1-\beta)S}{n^{2.5}}\right)^{1-\epsilon} + \left(\frac{\beta\gamma(1-\alpha)S}{n^{2.5}}\right)^{1-\epsilon}\right) \\ &\leq n^{2.5}\left(\frac{S}{n^{2.5}}\right)^{1-\epsilon}\left((1-\gamma)^{1-\epsilon} + (\gamma(1-\beta))^{1-\epsilon} + (\beta\gamma(1-\alpha))^{1-\epsilon}\right). \end{aligned}$$

We are going to show that the value of $F = (1-\gamma)^{1-\epsilon} + (\gamma(1-\beta))^{1-\epsilon} + (\beta\gamma(1-\alpha))^{1-\epsilon}$ is always no greater than 1 under our setting of parameters. First by $(1-x)^c \leq 1 - cx$ whenever $0 < c, x < 1$, we have

$$\begin{aligned} F &\leq (1-(1-\epsilon)\gamma) + \gamma^{1-\epsilon}\left(1-(1-\epsilon)\beta\right) + (\beta\gamma)^{1-\epsilon}\left(1-(1-\epsilon)\alpha\right) \\ &= 1 + \gamma\left(-(1-\epsilon) + \gamma^{-\epsilon}\left(1 + \beta\left(-(1-\epsilon) + \beta^{-\epsilon}\left(1-(1-\epsilon)\alpha\right)\right)\right)\right). \end{aligned}$$

By the condition that $\alpha > 10\epsilon(1+\log\frac{1}{\beta})$ and $\alpha, \beta \leq 1, \epsilon > 0$, we have $0 \leq \epsilon\log\frac{1}{\beta} < 1/10$, and in particular, $\epsilon < 1/10$. By the former and the fact that $e^x \leq 1 + 2x$ whenever $0 \leq x < 1/10$, we have $\beta^{-\epsilon} \leq 1 + 2\epsilon\log\frac{1}{\beta}$. By the latter, we have $1-(1-\epsilon)\alpha \leq 1 - 2\alpha/3$. Applying these two inequalities to the most inner term of F, and by the fact that all coefficients being multiplied to it are nonnegative, we have

$$\begin{aligned} F &\leq 1 + \gamma\left(-(1-\epsilon) + \gamma^{-\epsilon}\left(1 + \beta\left(-(1-\epsilon) + (1+2\epsilon\log\frac{1}{\beta})(1-\frac{2\alpha}{3})\right)\right)\right) \\ &\leq 1 + \gamma\left(-(1-\epsilon) + \gamma^{-\epsilon}\left(1 + \beta\left(\epsilon + 2\epsilon\log\frac{1}{\beta} - \frac{2\alpha}{3}\right)\right)\right) \\ &\leq 1 + \gamma\left(-(1-\epsilon) + \gamma^{-\epsilon}\left(1 + 2\beta\epsilon(1+\log\frac{1}{\beta}) - \frac{2\alpha\beta}{3}\right)\right) \\ &\leq 1 + \gamma\left(-(1-\epsilon) + \gamma^{-\epsilon}(1 + \frac{2\alpha\beta}{10} - \frac{2\alpha\beta}{3})\right) \\ &\leq 1 + \gamma\left(-(1-\epsilon) + \gamma^{-\epsilon}(1 - \frac{\alpha\beta}{3})\right). \end{aligned}$$

By $\alpha\beta > 10\epsilon(1+\log\frac{1}{\gamma})$ and the above argument, we have $\gamma^{-\epsilon} \leq 1 + 2\epsilon\log\frac{1}{\gamma}$. Applying it to F, we have

$$\begin{aligned} F &\leq 1 + \gamma\left(-(1-\epsilon) + (1+2\epsilon\log\frac{1}{\gamma})(1-\frac{\alpha\beta}{3})\right) \\ &\leq 1 + \gamma\left(\epsilon + 2\epsilon\log\frac{1}{\gamma} - \frac{\alpha\beta}{3}\right) \\ &\leq 1 + 2\gamma\epsilon(1+\log\frac{1}{\gamma}) - \frac{\alpha\beta\gamma}{3} \\ &\leq 1 + \frac{2\alpha\beta\gamma}{10} - \frac{\alpha\beta\gamma}{3} \\ &\leq 1 - \frac{\alpha\beta\gamma}{10} \leq 1. \end{aligned}$$

This proves $T(S) \leq n^{2.5}\left(\frac{S}{n^{2.5}}\right)^{1-\epsilon}$, in particular, $T(n^3) \leq n^{3-\epsilon/2}$. Therefore, summing up the two cases, the total running time of the algorithm is at most $O(cn^3 + n^{3-\epsilon/2})$.

□

Remark. The algorithm in Section 2 has parameters $c = \frac{1}{\Delta^4}, \alpha = 1, \beta = \frac{1}{\Delta}, \gamma = \frac{1}{\Delta^2}, \epsilon = \Theta(\frac{1}{\Delta \log \Delta})$, while G' is the subgraph induced by (A, B_1, C_1) or the entire graph if all vertices in A have low degree.

4 Conclusion

We have shown how to generalize the idea of divide-and-conquer in Chan's algorithm, and have provided a more intuitive proof of the recursion. The way of analysing the "sublinear" recurrence in Theorem 1, i.e., our $T(S)$ and its analysis, should be able to extend to other problems. We would like to see more applications of this method in proving the efficiency of other combinatorial algorithms that are based on divide-and-conquer.

Also, we would hope to have an $O(n^2)$ time algorithm for triangle detection on tripartite graphs with vertex set sizes n, n, and $\hat{O}(\log^4 n)$. We call this the "lopsided" triangle detection problem, where one side of vertices is very small compared to the others. An argument similar to the reduction by Vassilevska Williams and Williams [13] shows that this would yield an $O(n^2)$ time algorithm for multiplying $n \times n$ and $n \times \hat{O}(\log^4 n)$ Boolean matrices, improving the maximum outer dimension d that $n \times n$ and $n \times d$ Boolean matrices can be multiplied in $O(n^2)$ time, in both the combinatorial and the algebraic world. The current record of d is $\hat{O}(\log^3 n)$ by Chan's algorithm (Chan gave an $O(n^2)$ time algorithm for multiplying $n \times d$ and $d \times n$ matrices, which implies an $O(n^2)$ time triangle finding algorithm), while the record in the algebraic world is merely $O(\log n)$ [3].

Finally, we provide one type of instance for the lopsided triangle detection problem which seems hard to solve in $O(n^2)$ time with our current techniques: a graph G on vertex sets A, B, C with $|A| = |B| = n$ and $|C| = \hat{O}(\log^4 n)$, with roughly $1/\log n$ fraction of edges between A and C, $1/\log n$ fraction of edges between B and C, and constant fraction of edges between A and B. The main difficulty is that the size of C is too small. If we try to do recursion, its size will reach a constant too soon. Once it becomes of constant size, we basically have no way to save anything from exhaustive search. Trying to solve this type of instance might be a good starting point towards the goal of multiplying $n \times n$ and $n \times \hat{O}(\log^4 n)$ Boolean matrices in $O(n^2)$ time, which is an improvement even if algebraic algorithms are allowed.

Acknowledgement. The author would like to thank Ryan Williams for helpful discussions on results and writing of the paper, and the anonymous reviewers for their valuable comments.

References

[1] V. Z. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzhev. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics Doklady*, 11(5):1209–1210, 1970.

[2] Nikhil Bansal and Ryan Williams. Regularity lemmas and combinatorial algorithms. In *50th Annual IEEE Symposium on Foundations of Computer Science*, pages 745–754, 2009.

[3] R. W. Brockett and D. Dobkin. On the number of multiplications required for matrix multiplication. *SIAM Journal on Computing*, 5:624–628, 1976.

[4] Timothy M. Chan. Speeding up the four russians algorithm by about one more logarithmic factor. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 212–217, 2015.

[5] Michael J Fischer and Albert R Meyer. Boolean matrix multiplication and transitive closure. In *12th Annual Symposium on Switching and Automata Theory*, pages 129–131, 1971.

[6] Alan M. Frieze and Ravi Kannan. Quick approximation to matrices and applications. *Combinatorica*, 19(2):175–220, 1999.

[7] M. E. Furman. Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. *Soviet Mathematics Doklady*, 11(5):1252, 1970.

[8] Russell Impagliazzo, Shachar Lovett, Ramamohan Paturi, and Stefan Schneider. 0-1 integer linear programming with a linear number of constraints. *CoRR*, abs/1401.5512, 2014.

[9] François Le Gall. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation*, pages 296–303, 2014.

[10] Ian Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[11] Leslie G. Valiant. General context-free recognition in less than cubic time. *Journal of computer and system sciences*, 10(2):308–315, 1975.

[12] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the 44th Symposium on Theory of Computing Conference*, pages 887–898, 2012.

[13] Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *51th Annual IEEE Symposium on Foundations of Computer Science*, pages 645–654, 2010.