Beating Trivial Time for Tricky Triangle Tasks

Neha Pant
MIT, Cambridge, MA, USA

Ryan Williams
MIT, Cambridge, MA, USA

Abstract

For several well-studied triangle detection problems in the literature, the trivial enumeration algorithms are known to be optimal (up to the exponent) assuming popular fine-grained conjectures. For example, All-Edges Sparse Triangle and Sparse Monochromatic Triangle where each node has degree n^δ for some $\delta < 1$, and the Exact Triangle where edges have arbitrary weights, all have this property under the 3SUM Conjecture. However, as there are slightly nontrivial algorithms for 3SUM, it is natural to wonder if the trivial algorithm for these tricky triangle tasks might also be improved.

Applying a variety of techniques from randomized algorithms, circuit complexity, and communication complexity, we present the first improvements over the trivial algorithms for each of these problems in the Word RAM model. Moreover, our algorithms can be implemented with *only* polysize AC0 operations on words. Extending our techniques, we also show how to solve the notorious 4-cycle detection problem on n-node graphs in $o(n^2)$ time, in a Word-RAM model with word size $w > \omega(\log^2 n)$. Along the way, we show how to sort n items over a universe of size 2^u using only AC0 word operations in $O(nu \log n)/w$ time.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases sparse graph algorithms, triangle, Word RAM, 4-cycle

Digital Object Identifier 10.4230/LIPIcs.MFCS.2026.14

Related Version *Full Version*: http://arxiv.org/abs/2606.27727

Funding Parts of this work were completed while R.W. was visiting the Institute for Advanced Study, Princeton, NJ. This material is based upon work supported by the National Science Foundation under grants DMS-2424441 (at IAS) and CCF-2420092 (at MIT).

1 Introduction

Arguably the most fundamental graph problems in fine-grained complexity are concerned with finding small cliques of various kinds, especially triangles (3-cliques). Among the large class of problems that are known to be *subcubic equivalent* to the All-Pairs Shortest Paths (APSP) problem, the most basic such problem is Negative Triangle: find a triangle with negative edge sum in an edge-weighted graph [37]. Both the quadratic-time hardness of the infamous 3SUM problem and the cubic-time hardness of the APSP problem imply several hardness hypotheses for triangle problems:

- the hypothesis that the Exact Triangle problem, of finding a triangle with edge sum zero [38] in an edge-weighted graph, requires $n^{3-o(1)}$ time, and
- the hypothesis that the All-Edges Sparse Triangle problem, of determining whether each edge in an n-node graph with max degree at most n^δ participates in a triangle, requires $n^{1+2\delta-o(1)}$ time [39].

Similarly, for the All-Edges Sparse Monochromatic Triangle problem, where we have an n-node graph with colors on its edges and maximum degree at most n^δ and we want to determine which edges participate in monochromatic triangles, it is known that the problem requires $n^{1+2\delta-o(1)}$ time under both the APSP and the 3SUM hypotheses [31, 39].

Already from the work of Fredman in 1976 [24] leading up to the work of Williams [40] on APSP, it has been known that Negative Triangle can be solved in $o(n^3)$ time on n-node graphs, with the best known running time being $n^3/2^{\Omega(\sqrt{\log n})}$ [40]. In contrast, there have been no reported improvements over the obvious running times for All-Edges Sparse Triangle, Exact Triangle, and All-Edges Sparse Monochromatic Triangle. In this paper, we present improved algorithms for all three of these problems.

Our Results for Triangles in Sparse Graphs. For a parameter $\delta \in (0, 1/2]$, the All-Edges Sparse Triangle problem asks: *given an n-node graph where each node has degree at most n^δ, decide whether each edge of the graph is in a triangle.*

Although for dense graphs, it is well-known that matrix multiplication leads to faster algorithms [6], the best known algorithm for All-Edges Sparse Triangle simply enumerates all pairs of neighbors from each node, taking $O(n^{1+2\delta})$ time. Under standard fine-grained hardness assumptions, there is a matching lower bound. The work of Pătraşcu [32] showed that the problem requires $n^{1+2\delta-o(1)}$ time under the 3SUM hypothesis. Vassilevska W. and Xu [39] showed the same lower bound holds under the Exact Triangle Hypothesis (the hypothesis being that the problem requires $n^{3-o(1)}$ time). We note that this reduction incurs an $O(\log n)$ factor, and therefore shaving a log factor from All-Edges Sparse Triangle does not immediately carry over to Exact Triangle. Chan, Vassilevska W. and Xu [14] gave a similar conditional lower bound assuming that either APSP or 3SUM over the reals cannot be solved in $n^{3-\varepsilon}$ time or $n^{2-\varepsilon}$ time, respectively.

We present the first improvement over the trivial algorithm for the problem. To keep ourselves honest (disallowing arbitrary multiplications and disallowing arbitrary lookup tables of size 2^w), our computational model will be the Word RAM model with $\mathsf{AC0}$ operations on words of w bits. That is, our unit-time operations are restricted to those that computed on a constant number of w-bit words, by a $\mathsf{poly}(w)$-size $\mathsf{AC0}$ circuit. (Note $w = \Theta(\log n)$ is the standard choice.) This is a standard model; see Section 2 for more details.

Please observe we are making a *restriction* on the standard Word RAM model. When $w \leq \delta \log n$ for $\delta < 1$, our algorithms can be implemented with lookup tables for practically any reasonable random-access model with constant-time lookup. Our theorems imply not only that all these problems can be solved faster in the standard Word RAM, *but also* in a model with a more restricted set of operations where no 2^w-bit lookup tables are allowed (which would permit *arbitrary* operations on w-bit words).

▶ **Theorem 1.** *For all $\delta \in (0, 1/2]$, there is a Las Vegas algorithm for* All-Edges Sparse Triangle *on graphs with maximum degree n^δ running in $O(n^{1+2\delta} \log w)/w$ expected time, where w is the word size.*

Our ideas are general enough that they extend in a natural way to solve the apparently harder *monochromatic* triangle problem in sparse graphs as well.

▶ **Theorem 2.** *For all $\delta \in (0, 1/2]$, there is a Las Vegas algorithm for* All-Edges Sparse Monochromatic Triangle *running in $O(n^{1+2\delta} \log w)/w$ expected time.*

Our techniques imply a new data structure for the Set Intersection query problem: given sets $S_1, \ldots, S_n$ each of size $O(d)$, we wish to preprocess them so that given a query $(i, j) \in [n] \times [n]$, we can determine if $S_i \cap S_j = \varnothing$ as fast as possible. There has been a line of work on this fundamental problem. Among other results, Bille, Pagh, and Pagh [10] show how to compute the intersection of m sets with n total elements in time $O(n(\log w)^2/w + km)$ time, where k is the size of the intersection. Kopelowitz, Pettie, and Porat [29] give a dynamic data

structure (supporting insertions and deletions in $O(1)$ time). When all sets have cardinality at most d, their data structure can answer set intersection queries in $O(d \log^2 w)/w$ time. They use two levels of pairwise independent hashing to pack elements into words, using a merging routine of Albers and Hagerup [4] to compute intersections. Eppstein, Goodrich, Mitzenmacher, and Torres [20] showed how to remove a $\log w$ factor from these results. In particular, they presented a data structure using cuckoo hash tables and cuckoo filters to determine the intersection of two sets of total size d in $O(d \log w)/w$ expected time. The behavior of their data structure is rather complex, requiring a sophisticated analysis of the "cyclomatic number" of random 3-uniform hypergraphs. They also assume *completely random* hash functions (which are necessary in order to prove their tail bounds, see [20] p. 249, bullet 1) which are effectively random lookup tables. In contrast, our algorithms use only simple (linear and $O(\log n)$-wise independent) hashing and AC0 word operations, requiring very little randomness (at most polylogarithmic in n). To summarize:

▶ **Corollary 3.** *There is a data structure for the* Set Intersection *query problem with $\tilde{O}(nkw)$ preprocessing time, such that all queries can be answered in $O(1 + (k \log w)/w)$ time, using only* poly$(\log n)$ *randomness in preprocessing and* AC0 *operations on w-bit words.*

As in [29, 20], other algorithmic applications follow such as improved triangle listing algorithms; see these papers for references.

Our Results for Exact Triangle. The Exact Triangle problem has been widely studied as the basis for a variety of conditional lower bounds (for instance, [38, 26, 37, 39, 28, 15]). In particular, the hypothesis that the problem requires $n^{3-o(1)}$ time on n-node graphs is widely believed, as a faster algorithm would simultaneously refute multiple hardness conjectures in fine-grained complexity. We show how to shave a $\log^{3/2} n$ factor from the n^3 time bound in standard models, and a $w^{3/2}$ factor for the w-bit Word RAM.

▶ **Theorem 4.** *There is a Las Vegas algorithm for* Exact Triangle *on n-node graphs in $O(n^3 (\log w)^{3/2}/w^{3/2})$ expected time.*

An Attack on the 4-Cycle Problem. Finally, we consider the notorious problem of detecting a 4-cycle in an undirected graph on n nodes and m edges. An $O(n^2)$-time algorithm is known [33] and an $O(m^{4/3})$-time algorithm is known [6], but these running times coincide when the number of edges is $\Theta(n^{1.5})$. Indeed, the "hard case" of 4-cycle detection is when the graph has $O(n^{1.5})$ edges: Bondy and Simonovits showed that any graph with $cn^{1.5}$ edges for sufficiently large constant c must have a 4-cycle [11], and works due to Brown and Erdős, Rényi, and Sós gave algebraic constructions of 4-cycle free graphs that match this bound [12, 21]. There are essentially no hardness results known for the detection version of this problem, and yet no algorithmic improvements are known beyond the $O(n^2)$-time solution, which enumerates pairs of neighbors of nodes and looks for a collision. We note that by contrast, much progress has been made on 4-cycle listing. In particular, there is a $O(\min(n^2, m^{4/3}) + t)$ time algorithm [2], where t is the number of 4-cycles, along with a matching lower bound under the 3SUM hypothesis [1, 28].

We give two results on 4-cycle detection. First, building on our ideas for finding triangles in sparse graphs, we reduce 4-cycle detection to a variant of the Element Distinctness problem.

▶ **Definition 5** (Concatenated Element Distinctness)**.** *We are given k lists $L_1, \ldots, L_k$, where each list has n elements of ℓ-bit strings. For every $i, j \in [k]$, element-wise concatenate them to a produce a new list $L_{i,j}$ containing n elements, each of which now have 2ℓ bits. The task is to decide whether or not for all $i \neq j$, all n strings in $L_{i,j}$ are distinct.*

In other words, the k lists form $O(k^2)$ instances of Element Distinctness on n elements, and we accept if and only if all of the generated instances have distinct elements.

▶ **Theorem 6.** *For $T(n,k,\ell) \geq n + k + \ell$, if Concatenated Element Distinctness can be solved in $O(T(n,k,\ell))$ time, then 4-cycle detection in n-node graphs can be done in $O(T(n,\sqrt{n},(\log_2 n)/2))$ time.*

Motivated by this reduction, we show how to sort n items over a universe of size 2^u in $O(nu \log n)/w$ time in the AC0 Word RAM model on w-bit words. Prior work either required more complex word operations such as 2^w-size lookup tables [13], or extra logarithmic factors [4]. As sorting easily solves Element Distinctness, we obtain the following:

▶ **Theorem 7.** *We can detect a 4-cycle in n-node graphs in $O(n^2 \log^2 n)/w$ time in the AC0 Word RAM.*

For word size $w \leq \log^2 n$, this result is suboptimal, as we already know an $O(n^2)$-time algorithm when $w = \Theta(\log n)$. Nevertheless, it was not obvious how to improve over $O(n^2)$ time even for large word sizes: the known algorithms involve making $O(n^2)$ unstructured probes into a table, enumerating all pairs of neighbors of a given node to find a common pair of neighbors. We believe the reduction of Theorem 6 may be useful for future work.

Sublinear Time 4-Cycle Detection in Denser Graphs. Finally, we observe that 4-cycles can be found surprisingly quickly in graphs that have slightly more than $n^{1.5}$ edges. In fact, they can be found in *sublinear time* in the number of edges.

▶ **Theorem 8.** *There is a Las Vegas algorithm that, given a graph with n vertices and at least $n^{3/2+\delta}$ edges for $\delta \in (0, \frac{1}{2})$, returns a 4-cycle and runs in expected $O(n^{5/4-\delta/2})$ time.*

Therefore, on graphs with $\varepsilon n^{3/2}$ edges where $\varepsilon > 0$ is a small constant, the best known method for detecting (and finding) a 4-cycle remains $O(n^2)$ time. However, on graphs with $\varepsilon n^{3/2+\delta}$ edges for any positive δ, the time complexity of 4-cycle surprisingly drops to $O(n^{5/4-\delta/2})$ time.

A classic supersaturation result due to Erdős and Simonovits [22] states that a graph with $m \gg n^{3/2}$ edges has at least $\Omega((m/n)^4)$ 4-cycles, so there is a natural *random sampling* algorithm for finding 4-cycles: simply try uniform random 4-tuples of nodes! Our algorithm outperforms the random sampling algorithm when the number of edges is not significantly larger than $n^{3/2}$. For instance, when $m = \Theta(n^{3/2+\delta})$, supersaturation guarantees $\Omega(n^{2+4\delta})$ 4-cycles, so random sampling takes expected $O(n^{2-4\delta})$ time. This is strictly slower than our method when $\delta \in \left(0, \frac{3}{14}\right)$. Combining the two approaches, we obtain:

▶ **Corollary 9.** *There is a Las Vegas algorithm that, given a graph with at least $n^{3/2+\delta}$ edges for $\delta \in (0, \frac{1}{2})$, returns a 4-cycle and runs in expected $O(n^{\min\{5/4-\delta/2, 2-4\delta\}})$ time.*

Note: some proofs are omitted due to space constraints.

Intuition

Let us give some intuition for the new ideas and techniques that go into our results, starting with our results for triangles in sparse graphs. Fundamentally, triangle finding in sparse graphs amounts to computing a series of intersections of sparse sets. While there is a considerable literature on data structures for set intersection (for example [10, 17, 19, 29, 20]), prior work has focused on complex hashing schemes with complex word operations.

Our approach reduces triangle problems in sparse graphs to problems which are more easily "word-packable", and can be cleanly implemented in simple (AC0) word operations. For example, in Theorem 20 we show how to randomly embed set intersections of sets of size k into the *equality inner product problem*, in which we are given vectors u, v of length $O(k)$ with entries in $[n]$, and we wish to find a component i such that $u_i = v_i$. Since equality inner products are easy to pack into word operations, this already suffices to detect triangle in sparse graphs. To solve the all-edges case, we partition the universe of n nodes into $O(n^\delta/\log n)$ components using $O(\log n)$-wise independent hashing, and argue that each vertex neighborhood has $O(\log n)$ nodes in each component. We can then compute the $O(\log n)$-set intersections quickly by adapting a randomized communication protocol for set intersection [18]. We use similar ideas to get our reduction from 4-cycle to Concatenated Element Distinctness.

For the Exact Triangle problem, we can do similar tricks. However we can obtain an improved speedup (from a $\log n$ factor to a $\log^{3/2} n$ factor) by exploiting the density of the graph. In particular, we can pack small $O(\sqrt{\log n})$-size subgraphs of the graph in our words, along with linear hash values of weights.

In the process of proving our results on 4-cycle, we present an improved "packed sorting" algorithm. In this problem, we wish to sort a list of n items from the universe $[\mathsf{poly}(n)]$ where the unsorted list is provided in $(n \log n)/w$ words, each word containing $O(w/\log n)$ items. Although it was known how to sort a packed list in $O(n \log^2 n)/w$ time using arbitrary word operations, we show how to do it using only AC0 word operations. The key is that, once the sublists within each word are sorted themselves, all that remains is to merge the sublists. We show how to perform merges in AC0, by exploiting the sorted order to determine in parallel the ranks of all items in the merge.

Although our results only give "minor" running time improvements (polylog-shaving and log-log shaving) over the prior best-known algorithms, we achieve these results with *simple* techniques, by viewing these fundamental and well-studied problems in a different light. It is reasonable to believe that applying more sophisticated methods could lead to stronger improvements.

2 Preliminaries

Model of Computation. We use the AC0 Word RAM model of computation with word size w. This means that we *restrict* ourselves to word operations that can be computed by a $\mathsf{poly}(w)$-size circuit of constant depth with AND and OR gates of unbounded fan-in as well as NOT gates, and assume that any such operation can be computed in constant time [27, 7, 8]. Most standard Word RAM instructions can be implemented in the AC0 Word RAM, including addition and subtraction modulo 2^w, bitwise boolean operations, comparisons, and shifts [35, 16]. The only standard operation not supported in this model is multiplication. For wordsize $w \gg \log n$, we explicitly disallow 2^w-size lookup tables, which would allow us to compute *any* function on w bits: we only allow AC0 functions to be computed on w-bit words in constant time.

Graphs. Unless otherwise stated, all graphs are undirected and unweighted.

▶ **Theorem 10** (Reiman). *If G is a four-cycle free graph on n nodes, it has at most $\frac{n}{4}\left(1+\sqrt{4n-3}\right)$ edges.*

Chernoff lower bounds for k-wise independence. We will also use the following adaptation of Chernoff bounds to k-wise independent random variables:

▶ **Theorem 11** ([34]). *Let k be a positive integer, and let X be the sum of $0/1$ valued k-wise independent random variables, each with expectation μ. For $\delta \geq 1$ and $k \leq \lfloor \delta\mu/e^{1/3} \rfloor$, we have $\Pr[X \geq (1+\delta)\mu] \leq e^{-k/2}$.*

3 Compressed List Operations in the AC0-RAM

At a high level, our algorithms use various hashing schemes to reduce triangle and 4-cycle to problems on compressed lists, in particular, equality product, set disjointness, and element distinctness. We then take advantage of word-level parallelism and known techniques for designing AC0 circuits to solve these problems faster, giving us word size factor savings. We begin by detailing these compressed list operations.

▶ **Definition 12** (Compressed list representation). *Let L be a list of k elements from $[2^\ell]$, where $\ell \leq w$. We store such L in* compressed list representation *by packing $\lfloor w/\ell \rfloor$ contiguous list elements into a single word, for an overall packed length of $O(k\ell/w)$ words.*

In the following, we let $\perp$ denote a special character that does not match any valid element. This can be easily implemented in various ways; e.g., we can extend the ℓ-bit encoding of every string to $\ell + 1$ bits, where all "valid" ℓ-bit elements start with a zero, and $\perp$ equals all-ones (for example).

3.1 Sorting Compressed Lists

We begin by describing a sorting algorithm on compressed lists with AC0 word operations, loosely inspired by a work of Chan [13] who allowed *arbitrary* word operations. Given two compressed sorted lists L_0 and L_1, we define the *merge step of L_0 and L_1* to be the task of implementing the standard linear-time merge of L_0 and L_1 until one of the lists L_b is empty. The output of a merge step is the merged list obtained so far, along with a $b \in \{0, 1\}$ and the remaining content of the non-empty list L_b. We give a procedure to perform this operation on single words in compressed list representation, which we will use as a subroutine in the overall merge. In particular, at every step of the high-level merge, we will run the single-word merge on the leftover word from L_b with the next full word from $L_{\overline{b}}$.

▶ **Lemma 13.** *Let w be the word size. Given compressed sorted lists L_0 and L_1 where both lists are stored in a single word, the merge step of L_0 and L_1 can be implemented in constant time with AC0 word operations. We assume list elements are in the universe $[2^u]$ for $u \leq w$.*

Proof of Lemma 13. Since L_0 and L_1 are stored in a single word, we may assume there are at most $k = \lfloor w/u \rfloor$ elements in each. If the lists have fewer than k elements, we expect the entries to appear packed consecutively in the upper bits of the word, and the remainder of the word to be padded with u-bit $\perp$ entries which are treated as "infinity", larger than all other entries.

To sort $L_0 \cup L_1$, observe that it suffices to determine, for every element x in L_0, the number of elements in L_1 that precede x in sorted order, and for every element x' in L_1, the number of elements in L_0 that precede x' in sorted order. Using an AC0 circuit, exploiting the fact that L_0 and L_1 are already sorted, we can check all possible values for these numbers in parallel. We will perform comparisons as if all entries in L_0 have a trailing 0 appended to them, and all entries in L_1 have a trailing 1. This means that there does not exist any element that appears in both L_0 and L_1, which has the effect of breaking ties in favor of L_0.

To compute the kth value of the merged output, we check the following. For every $i \in [|L_0|]$, compute $j = k - i$. Then, check the following conditions:
(1) $L_0[i] > L_1[j]$ and
(2) either $L_0[i] < L_1[j+1]$ or $L_1[j] = \perp$.
If the AND of (1) and (2) holds, we output $L_0[i]$, else output 0. We then OR together all of these gadgets for all $i \in [|L_0|]$, and also OR these results with the analogous computation taking $i \in [|L_1|]$. If the condition is satisfied (say for $i \in [|L_0|]$), this means that there are $j+1$ elements in L_1 that are strictly smaller than $L_0[i]$, and the rest of the elements are strictly larger. Since elements in L_0 cannot be equal to those in L_1, there is only one such value of j per element. Additionally, since we enforce that our merge step is stable, the (unique) correct location in the merged list for this element is $i + j$, as desired.

As mentioned in Section 2, the addition and comparisons can be done in AC0. By construction, at most one comparison gadget fires a nonzero index, so we can simply OR them all. Thus, we can run these checks for all $|L_0|+|L_1|$ list elements in parallel, constructing our merged list index by index.

All that remains is to determine which list will be left empty following the merge step. To do this, let i and j be the last elements of L_0 and L_1. If $i < j$, then L_0 is the newly emptied list ($b = 0$), otherwise L_1 is emptied ($b = 1$). Let m be the minimum of i and j. We place all elements whose sorted rank is in $[m]$ into at most two words, which is the merged portion. We can pad the second word with $\perp$. We place the remaining non-$\perp$ elements (all from list $L_{\overline{b}}$) into another word (again padding with $\perp$), and return b, completing the merge step.

As mentioned above, each of these gadgets is constant-depth, and we compose them in parallel. For every element, we perform at most $\max(|L_0|, |L_1|)$ comparisons, so the circuit size is at most quadratic in the input size, and thus the entire construction is in AC0. ◀

▷ Claim 14. Given compressed sorted lists L_0 and L_1 with u-bit entries and at most k elements each, a merge of L_0 and L_1 can be implemented in $O(ku/w)$ time with AC0 word operations.

Now that we have a merge step, it is natural to construct a mergesort-like sorting algorithm. We note that this gives an improvement over the best-known packed sorting algorithm in the Word-RAM model, which is due to Belazzougui, Brodal, and Nielsen [9]. Their algorithm sorts n integers on w/u bits packed into compressed lists in $O(\frac{n}{u}(\log n + \log^2 u))$ time, and works for any choice of $w = \Omega(\log n)$. In particular, they are interested in the setting where each element is on $u = O(\log n)$ bits, so $u = O(w/\log n)$. Thus, their overall runtime is $O(\frac{n \log n}{w}(\log n + \log^2 w))$, which we will slightly improve.

▷ Claim 15. Given a list L of n elements in $[2^u]$ packed into words, we can sort L in $O(nu \log n/w)$ time.

Proof of Claim 15. We are able to compute the result of one AC0 circuit that takes w bits as input in constant time. Thus, in $\log w$ time, we have access to circuits with unbounded fan-in and logarithmic depth, which captures at least all of NC1, which, crucially, contains sorting [3]. So we require $O(\log w)$ time to sort the elements in a single word.

This serves as the preprocessing for our merging procedure, and runs in $O(n \log w/w)$ time. The bulk of the computation is the merge step. We can now merge each of the $n \log n/w$ words in a tree-like fashion. One can think of placing each word at the leaf of a height-$O(\log n)$ binary tree, and then storing the merged version of the children in every parent node. The recurrence associated with this is $T(m) = 2T(m/2) + O(m)$, so we require $O(m \log m)$ time, where $m := nu/w$ is the total number of words. Thus the overall runtime is $O(nu \log n/w)$, which dominates the preprocessing. ◁

We note that the difficulty of packed sorting may be inherent. In particular, if one could remove the $\log n$-factor from the runtime above, this would resolve the longstanding open problem of linear-time sorting for (almost) all word sizes. The Belazzougui, Brodal, and Nielsen paper achieves linear-time for $w = \Omega(\log^2 n \log\log n)$, leaving the problem open only for word sizes that are $\omega(\log n)$ (from radix sort) and $o(\log^2 n \log\log n)$. However, the paper does this by reducing sorting on word size w to $\log\log n$ levels of packed sorting of $O(n)$ elements on $O(\log n)$ bits. Thus, shaving off the log factor on the packed sorting runtime would give a $O(n \log n \log\log n/w)$ time algorithm, which is linear when $w = \Omega(\log n \log\log n)$, almost completely closing this longstanding gap in word size.

It is also known that shaving this log-factor, or more generally, solving packed sorting in time faster than $o(\min\{n, \frac{nu}{w} \log \frac{nu}{w}\})$, would refute the Undirected k-Pairs Conjecture, which is central to the field of network coding [30, 23].

3.2 Indicator Masks for Compressed Lists

Beyond sorting compressed lists, we would like to efficiently indicate list elements of interest to our algorithms. To do this, we define an *indicator mask* to be a length-w bitstring that is all zeros except the leading bit of each entry. This bit will indicate the value of some predicate evaluated on the associated entry in another word. We now show, inspired by bit tricks from [25], that we can produce a mask that indicates the all-zeros entries in $O(1)$ time.

▷ Claim 16. Let z be a word in a compressed list. There is a constant-time algorithm to produce an indicator mask whose ones indicate that the associated entries in z are all-zeros.

Our next task is to compute an indicator mask for set intersection.

▷ Claim 17. Given two sorted compressed lists L_0 and L_1 with k elements of length ℓ bits, which correspond to k-element sets S_0 and S_1, we can compute a mask that indicates elements in $S_0 \cap S_1$ in $O(k\ell/w)$ time.

The purpose of computing these indicator masks is to efficiently indicate "special" list entries that are of interest to our algorithms. However, these compressed lists are created as a result of hashing, and our algorithms generally need to filter out false positives by looking up the associated entries in the preimages. To that end, we need to be able to extract indices of the ones in our masks. We will use the following AC0 circuit to do this:

▷ Claim 18. In the AC0-RAM model, there is a constant-time algorithm that returns the index of the most significant set bit of a word.

Note that we can call this procedure, zero out the associated bit, and then repeat to extract the indices of all of the ones in a particular indicator mask. This only requires us to pay $O(1)$ time per set bit whose index we retrieve.

4 A Faster Algorithm for All-Edges Sparse Triangle

Armed with the compressed list operations from above, we are ready to solve triangle problems faster. We begin with a subquadratic algorithm for Sparse Triangle Detection, which is quite natural but is not strong enough to solve the All-Edges version faster. The goal here is to emphasize the relationship between triangle problems and equality product-like problems, and to motivate the reduction to set intersection that we will present in Section 4.2.

4.1 Warm-Up: Solving Sparse Triangle Detection Faster

We will solve sparse triangle detection by reducing to the following problem:

▶ **Definition 19** (All-Edges Equality Inner Product). *Let $\delta \in (0, 0.5]$ and $d = O(n^\delta)$. Given an $n \times d$ matrix A and an $d \times n$ matrix B with entries in $[n^{1-\delta}] \cup \{\perp\}$, as well as a set P of $O(n^{1+\delta})$ points in $[n] \times [n]$, is there an (i, j) in P and a value of $k \in [n^\delta]$ such that $v := A[i, k] = B[k, j]$ and $v \neq \perp$?*

▶ **Theorem 20.** *There is a $O(n^{1+\delta})$-time randomized reduction from the triangle detection problem on graphs of maximum degree $O(n^\delta)$ to All-Edges Equality Inner Product. When there is no triangle in the original graph, the All-Edges Equality Inner Product instance is a no instance. When there is a triangle, the All-Edges Equality Inner Product instance is a yes instance with constant nonzero probability.*

Proof of Theorem 20. We assume without loss of generality that G is tripartite, with parts I, J, and K each of size at most n. We begin by "color-coding" vertices as in [5], but with a large color palette. We impose the following rule: every vertex in $I \cup J$ has an edge to at most one vertex per color class of K. To enforce this, if a vertex $u \in (I \cup J)$ has multiple edges to one color class, we will select one uniformly at random to keep and delete the rest. We show that every triangle (u, v, w) in the graph survives with constant probability.

Suppose we have a pairwise independent hash family $\mathcal{H} = \{h : [n] \to [Cn^\delta]\}$ for large constant C, where each $h \in \mathcal{H}$ corresponds to a coloring of the vertices of K with Cn^δ colors. Now suppose $u \in I$, $v \in J$ and $w \in K$ form a triangle. Let $N(u)$ and $N(v)$ denote the neighbors of u and v in K, respectively. The probability that the triangle u, v, w survives is the probability that both (u, w) and (v, w) survive. Fix a color $c \in [Cn^\delta]$. We have:

$$\begin{aligned}
&\Pr[(u, v, w) \text{ survives and } h(w) = c] \\
&\quad = \Pr_{h \leftarrow \mathcal{H}}[h(w) = c \wedge (\forall k \in (N(u) \cup N(v)) \setminus \{w\})\ h(k) \neq c] \\
&\quad = \frac{1}{Cn^\delta} \Pr_{h \leftarrow \mathcal{H}}[(\forall k \in (N(u) \cup N(v)) \setminus \{w\})\ h(k) \neq c \mid h(w) = c] \\
&\quad \geq \frac{1}{Cn^\delta} \left(1 - \Pr_{h \leftarrow \mathcal{H}}[(\exists k \in (N(u) \cup N(v)) \setminus \{w\})\ h(k) = c \mid h(w) = c]\right) \\
&\quad \geq \frac{1}{Cn^\delta} \left(1 - |(N(u) \cup N(v)) \setminus \{w\}| \cdot \frac{1}{Cn^\delta}\right) \\
&\quad \geq \frac{1}{Cn^\delta} \left(1 - 2n^\delta \cdot \frac{1}{Cn^\delta}\right) \geq \frac{1}{Cn^\delta} \left(1 - \frac{2}{C}\right).
\end{aligned}$$

For each of the Cn^δ choices of c, all of the events in question are disjoint. Summing over all of them, the probability (u, v, w) survives is at least $1 - 2/C$. Setting $C \geq 1$ large, we conclude that every triangle in the original graph survives with constant probability.

We now use the color classes to express the remaining graph as a smaller matrix. We note that with probability at least $1 - 1/D^2$, all of the color classes have size at most $Dn^{1-\delta}$; in the end, we will choose D to be a large constant. To see this, let $X_c := \sum_{w \in K} \mathbb{1}[h(w) = k]$, the number of vertices that are assigned color class c. Note that $\mathbb{E}[X_c] = n^{1-\delta}/C$. By pairwise independence, $\mathsf{Var}(X_c) = \sum_{w \in K} \mathsf{Var}(\mathbb{1}[h(w) = k]) \leq n^{1-\delta}/C$. By Chebyshev's inequality,

$$\Pr[X_c - \mathbb{E}[X_c] \geq Dn^{1-\delta}] \leq \frac{1}{CD^2 n^{1-\delta}}.$$

Union bounding over all Cn^δ color classes, we have that all color classes have size at most $Dn^{1-\delta}$ with probability at least $1 - 1/D^2$. We construct an $n \times n^\delta$ matrix A that captures edges between I and K, as well as a $n^\delta \times n$ matrix B that captures edges between K and J.

Note that all indices i, j are in $[n]$ and index their respective vertex sets, whereas the k indices are in $[O(n^{1-\delta})]$ and determine a color class of K. We also assume that we re-index the nodes in each color class of K, so that the tuples $(k, v) \in [n^\delta] \times [O(n^{1-\delta})]$ uniquely identify some node in the kth color class of K.

We now set the (i, k) entry of A to the node in the kth color class of K that the vertex i has an edge to (and the (k, j) entry of B analogously). If the vertex has no edge to this color class, set the entry to $\perp$. Now, if there exists a k such that $A[i, k] = B[k, j] = v$ and $v \neq \perp$, both i and j share an edge to the vth node of the kth color class in K. Thus, if (i, j) is itself an edge, then $i, j, (k, v)$ form a triangle.

Thus, with constant probability, the All-Edges Equality Inner Product problem on the matrices A and B with $P := E$, the edge set of G, returns the same answer as the Sparse Triangle Detection problem on the original graph. ◀

We now give an algorithm to solve All-Edges Equality Inner Product faster than the obvious $n^{1+2\delta}$ time solution. The high-level idea is as follows: we use a linear hash function to hash the matrix entries down to $O(\log w)$ bits, write the problem down in a compressed matrix form by packing multiple entries into a single word, and then use word operations to compute the equality inner product.

▶ **Theorem 21.** *For every $\delta \in (0, 1/2]$ and word size $w \geq \Omega(\log n)$, there is a Las Vegas $O(n^{1+2\delta}(\log w)/w)$-time algorithm for the All-Edges Equality Inner Product problem on $n \times n^\delta$ and $n^\delta \times n$ matrices.*

While this algorithm works well for triangle *detection*, the color coding reduction preserves every triangle with constant probability. Thus, if we tried to use this approach to solve All-Edges Sparse Triangle, we would answer each edge query correctly with constant probability. However, we would like the following stronger correctness guarantee: the entire $O(n^{1+\delta})$-bit output should be correct with high probability. To do this, we need to circumvent the reduction to All-Edges Equality Inner Product.

4.2 Solving All-Edges Sparse Triangle in $O(n^{1+2\delta} \log w/w)$ Time

We begin by formally defining the All-Edges Sparse Triangle problem. We note that the instances that arise in the literature from the APSP and 3SUM reductions have $m = O(n^{1.5})$ and maximum degree $\Delta = O(\sqrt{n})$. We give a slight generalization, allowing for graphs of any sufficiently small sparsity. However, our techniques do require that the maximum degree $\Delta = O(m/n)$. We also assume without loss of generality that the graph is tripartite; if this is not the case, generate two more copies of the vertex set, and add the original edges of the graph between each of the pairs of parts.

▶ **Definition 22** (All-Edges Sparse Triangle)**.** *Let G be a tripartite graph whose vertex set is $I \cup J \cup K$ such that $|I| = |J| = |K| = n$. Let the number of edges be $m = n^{1+\delta}$ for some $\delta \in (0, 0.5]$, and let the maximum degree be $\Delta := O(n^\delta)$. For every edge $e \in E \cap (I \times J)$, decide whether e participates in a triangle.*

▶ **Reminder of Theorem 1.** There is a Las Vegas algorithm for All-Edges Sparse Triangle on graphs with maximum degree n^δ running in $O(n^{1+2\delta} \log w)/w$ expected time.

Proof of Theorem 1. Let the word size $w \geq c \log(n)$ for a constant $c > 0$. As in Section 4.1, we can think of solving All-Edges Sparse Triangle as responding to $O(n^{1+\delta})$ edge queries, each in $o(n^\delta)$ time. Our algorithm is loosely inspired by a communication complexity protocol for set disjointness due to Dasgupta, Kumar, and Sivakumar [18]. We do the following:

1. Randomly partition K into n^δ/w color classes.
2. Compress the name of each vertex in K using a random hash function: $h : [n] \to [\mathsf{poly}(\ell)]$.
3. For each vertex i in $I \cup J$, for every color class v in $[n^\delta/w]$, write down a compressed list that stores the hashes of their neighbors in that color class in sorted order.
4. To respond to an edge query (i, j), for every color class, compute the set intersection of the compressed lists associated with i and j's neighbors with that color. If it is non-empty, (i, j) participates in a triangle.

We will now discuss each step in more detail. We begin by color-coding the vertex set, randomly partitioning $[n]$ into n^δ/w color classes $S_1, \ldots, S_{n^\delta/w}$. We need to draw h from an $O(\log n)$-wise independent hash family, as follows.

Fix an arbitrary vertex $i \in (I \cup J)$ and color class $v \in [n^\delta/w]$. We define the random variable $X_{i,v} := |N(i) \cap S_v|$ to be the number of neighbors that u has in the vth color class. Clearly, $\mathbb{E}[X_{i,v}] = w \geq c \log(n)$. By the k'-wise independent Chernoff bound (Theorem 11), we know that for $\delta' \geq 1$ and $k' \leq \delta' \lfloor \mu/e^{1/3} \rfloor$ we have $\Pr[X \geq (1+\delta')\mu] \leq e^{-k'/2}$. Setting $k' = 6\ln(n)$ and setting $\delta' = 6e^{1/3}/c \leq O(1)$, we have

$$\Pr[X_{i,v} \geq (1+\delta')w] \leq 1/n^3.$$

We can union bound over all the color classes and all the vertices, which tells us that each vertex's neighborhood in each color class has size at most $k := (1+\delta')w$ with probability $1 - 1/n$. If this does not hold for all vertices, we can just rerun the color coding.

We will now run a second level of hashing. We pick a random hash function $h : [n] \to [k^3]$ such that each element in the image can be expressed in $\ell := 3\log w + O(1)$ bits. We use ℓ copies of the linear hash function $h_r(x) := \langle r, x \rangle \bmod 2$. One might (rightly) worry that computing an inner product modulo 2 is *not* an $\mathsf{AC0}$ operation. We sidestep this issue as follows: note that all of our hashing is done in the preprocessing step, and that we are only hashing $O(n^{1.5})$ values. So although the hashes technically cannot be computed in $\mathsf{AC0}$, we do not need to use word-level parallelism here: we can afford an extra w factor in the running time of preprocessing. Note we also need at most $O(\mathsf{poly}(\log n))$ bits of randomness to specify the hash function.

So for every vertex $i \in I \cup J$ and every color class S_v, we can hash the elements of $N(u) \cap S_v$, put them in sorted order, and write them down in compressed form using $k\ell/w$ words. Note that we can have collisions here, so we will store a mapping from the hashes in the image to the vertices in the preimage (written down in sorted order), which requires $O(n^{1+\delta}w)$ time. Again, we require $O(\mathsf{poly}(\log n))$ bits of randomness, all of which is used to build our word-packed lists in preprocessing.

When we receive an edge query (i, j), we need to determine for every color class S_v whether the associated sets $N(i) \cap S_v$ and $N(j) \cap S_v$ have nonempty intersection. If we fix a color class v and look at the associated packed lists, by Claim 17, we can produce an indicator mask corresponding to their intersection in time $O(\log w)$. However, since we have hashed the vertex names, it is possible that items in the intersection are false positives. Thus, we need extract the values in the intersection, look them up in the preimage, and verify that the original entries match as well.

We first bound the probability of false positives. The probability that any two entries collide is $1/k^3$, and there are $\binom{k}{2}$ pairs of entries, so we get at most $1/2k$ collisions per color class per query in expectation. We then look up the true vertex names in the preimage. To do this, we can think of these as being written down in sorted order, where each value takes up one word. We then merge the two lists and compute the true intersection in time linear

in the size of the preimage, which in expectation, is $O(1/k^2)$. We can also think of this cost as the number of collisions within the neighborhoods of i and j. So the expected cost of checking all of these collisions for a particular color class is $O(1/k)$, or linear in their number.

Recall that we have n^δ/w color classes, so computing the indicator masks over all of them requires time $O(n^\delta \log w/w)$ per query. As a result, we can afford to spend asymptotically that same amount of time checking false positives. Let P be the number of collisions we have to check. By construction, we have $\mathbb{E}[P] = \frac{1}{2k} \cdot \frac{n^\delta}{w} = \frac{n^\delta}{2(1+\delta')w^2}$. By a Markov bound,

$$\Pr\left[P \geq n^\delta \log w/w\right] \leq \frac{1}{2(1+\delta')w \log w} \leq \frac{1}{2}.$$

Thus, as in the All-Edges Equality algorithm in Theorem 21, if we see more than $n^\delta \log w/w$ collisions, we rerun the query in a subsequent round of the algorithm. The number of queries required is upper bounded by $n^{1+\delta} \cdot \sum_{i=0}^{\infty} 2^{-i} \leq 2n^{1+\delta}$. Thus, we require a total of $O(n^{1+\delta})$ queries, each of which takes $O(n^\delta \log w/w)$ time. As a result, we can compute All-Edges Sparse Triangle in expected time $O(n^{1+2\delta} \log w/w)$. ◀

5 Solving Other Triangle Problems Faster

We now look at other triangle problems whose algorithms can be improved using the techniques from above. We begin by looking at all-edges monochromatic triangle.

5.1 All-Edges Monochromatic Triangle Detection

Vassilevska, Williams, and Yuster gave a $O(n^{(3+\omega)/2})$ time algorithm for the detection version of the problem [36]. Thus, in the dense case, it seems to be meaningfully harder than triangle detection. However, in the sparse setting, we give an algorithm that runs in the same slightly subquadratic runtime as we get from Theorem 1 for All-Edges Sparse Triangle. We define the problem in the usual way, but require the additional stipulation of bounded degree.

▶ **Definition 23** (All-Edges Sparse Monochromatic Triangle)**.** *Let G be a tripartite graph whose vertex set is $I \cup J \cup K$ and $|I| = |J| = |K| = n$. Let the maximum degree in the graph be $O(n^\delta)$. The graph has colors along the edges; that is, there is a function $c : E \to [n^2]$. For every edge $e \in E \cap (I \times J)$, decide whether e participates in a monochromatic triangle.*

▶ **Reminder of Theorem 2.** There is a Las Vegas algorithm for the All-Edges Sparse Monochromatic Triangle problem that runs in expected time $O(n^{1+2\delta} \log w)/w$.

5.2 A Faster Algorithm for Exact Triangle

We now consider Exact Triangle, which, unlike the problems mentioned above, can be sped up even in the dense case. The problem is known to require $n^{3-o(1)}$ time under both the 3SUM hypothesis [38] and the APSP hypothesis [37], but it admits a log-factor speedup using an equality product based approach similar to the one in Section 4.1.

▶ **Definition 24** (Exact Triangle)**.** *This problem asks, given a target T and an n node graph with edge weights in $\{-n^c, \ldots, n^c\}$ for some integer $c > 0$, does there exist a triple of vertices p, q, r such that $w(p, q) + w(q, r) + w(r, p) = T$?*

▶ **Reminder of Theorem 4.** There is a Las Vegas algorithm that solves Exact Triangle on n-node graphs in expected $O(n^3 \log w/w^{3/2})$ time.

Proof of Theorem 4. Without loss of generality, we may assume the graph $G = (V, E)$ is tripartite with parts I, J, and K, where each part has $n/3$ nodes. Given a target weight T, and weight function $w' : E \to \mathbb{Z}$, we want to determine if there are $i \in I$, $j \in J$, and $k \in K$ such that i, j, k is a triangle and $w'(i, j) + w'(j, k) + w'(k, i) = T$. For simplicity, we assume that all weights are in $[-2^w, 2^w]$ where w is the word size.

Let $t = \sqrt{w}/(6\sqrt{\log w})$. We split the nodes of I, J, and K into at most $n/t + 1$ *groups* of t nodes each. Our idea is to check each triple of groups (containing $3t$ nodes and $O(t^2)$ edges) for an exact triangle, using a constant number of word operations.

To this end, we let $\ell := 10 \log w$, and we will hash all edge weights modulo a random prime p from the integer interval $\left[2, 2^\ell\right]$. (To be clear, we replace each edge weight with the corresponding residue in $\{0, 1, \ldots, p-1\}$.) By the Prime Number Theorem, a random integer in this range is prime with probability $\Omega(1/\ell)$. We can efficiently find such a prime by picking integers at random and primality testing. Note that this process costs only an (additive) $\mathsf{poly}(\log w)$ factor.

Now take some triple of groups $I' \subseteq I$, $J' \subseteq J$, $K' \subseteq K$, where $|I'| = |J'| = |K'| = t$. The number of bits needed to encode the weighted subgraph induced by $I' \cup J' \cup K'$ is

$$3t^2 \cdot \ell = 3\left(\frac{\sqrt{w}}{6\sqrt{\log w}}\right)^2 \cdot (10 \log w) < w,$$

so the entire subgraph can be packed into a single word.

Next, we observe that the problem of checking whether a given triple I', J', K' contains an exact triangle modulo p can be computed with a $\mathsf{poly}(w)$-size $\mathsf{AC0}$ circuit. Let $T_p = (T \bmod p)$ be the corresponding integer in $\{0, 1, \ldots, p-1\}$. Our circuit takes an OR over all $O(t^3) \leq O(w^{3/2})$ triples of nodes $i \in I'$, $j \in J'$, and $k \in K'$, then checks that the edges $(i, j), (j, k), (k, i)$ exist and

$$w'(i, j) + w'(j, k) + w'(k, i) \in \{T_p, p + T_p, 2p + T_p\}.$$

(Since only three weights are being summed, these are all the possible choices for the integer sum to equal the target modulo p.) Since addition is in $\mathsf{AC0}$, this construction is computable in $\mathsf{AC0}$. As a consequence, there is a single word operation to determine whether the triple I', J', K' contains an exact triangle mod p. For notational brevity, let's call this an *mod triangle check* which returns a yes or no answer.

All of the ingredients are now in place for us to describe the algorithm in full. Pick a uniform random prime p from $\left[2, 2^\ell\right]$, hash all edge weights modulo p, and for all $(n/t)^3$ triples I', J', K' on $3t$ nodes, we run a mod triangle check in constant time. For each check that returns yes, we enumerate all $O(t^3)$ triples of nodes $i \in I'$, $j \in J'$, $k \in K'$ to determine if there is an exact triangle in I', J', K' with the original unhashed weights (returning this triangle if it's found). If more than $(n/(tw))^3$ checks of triples return yes, then we stop and say *don't know*. Otherwise, we return *reject* if no exact triangle was found after enumerating over all triples I', J', K'.

First we verify the running time. By construction, our algorithm takes time $\mathsf{poly}(\log w) + O(n^2)$ to pick a random prime of $\ell = \Theta(\log w)$ bits and hash all weights to ℓ bits. Next, it takes $O((n/t)^3)$ time to call the mod triangle check on all triples I', J', K'. Finally, it takes $O((n/(tw))^3 \cdot t^3) \leq O((n/w)^3)$ extra time to go through up to $(n/(tw))^3$ triples and exhaustively check the corresponding $O(t)$-node subgraph for an exact triangle. Since $t < w$, the running time is bottlenecked by $O(n^3/t^3) \leq O(n^3(\log w)^{3/2}/w^{3/2})$.

Next, we show correctness. Note that if there is an exact triangle solution, then it is in some triple I', J', K', and the mod triangle check called on I', J', K' will always return *yes* no matter what prime is chosen. Thus the algorithm will find the exact triangle, *provided the algorithm does not stop early and return don't know.* Moreover, if the algorithm does not return *don't know* when there is no exact triangle, then the algorithm correctly rejects.

Finally, similarly to Theorem 21, we prove that the algorithm outputs *don't know* with probability at most $1/\mathsf{poly}(w)$. Hence with probability at least $1 - 1/\mathsf{poly}(w)$, the algorithm does not stop early: it will return an exact triangle when one exists, and will correctly reject when an exact triangle does not exist.

Fix some triple I', J', K' for which we will run a mod triangle check. For each $i \in I'$, $j \in J'$, and $k \in K'$, the probability that

$$w'(i,j) + w'(j,k) + w'(k,i) \neq t \quad \text{and} \quad w'(i,j) + w'(j,k) + w'(k,i) = t \bmod p$$

is at most the number of prime factors of $t - w'(i,j) + w'(j,k) + w'(k,i)$ divided by the total number of primes in $[2, 2^\ell]$, which is at most $2w \cdot (\ell/2^\ell) \leq (20 \log w)/w^9$. (Here, we apply the assumption that each edge weight fits in a word, in which case each weight has at most w prime factors, and their sum has at most $2w$ prime factors.) By a union bound over all t^3 triples of nodes in I', J', K', the probability that some $i, j, k \in I', J', K'$ do not form an exact triangle yet the mod triangle check returns yes on I', J', K' is at most $20t^3(\log w)/w^9$. As there are $(n/t)^3$ triples in total, the *expected* number of false-positive triples (there is no exact triangle yet the mod triangle check returns yes) is at most $20n^3(\log w)/w^9$. By Markov's inequality, the probability that the number of false-positive triples exceeds $n^3/(t^3w^3)$ is at most

$$20t^3(\log w)/w^6 < 20w^{3/2}(\log w)/w^6 < 20/w^4$$

for w sufficiently large. Therefore the probability the algorithm outputs *don't know* is at most $20/w^4$, which is $o(1)$. ◀

6 Attacks on 4-Cycle

We now turn to the problem of 4-cycle detection. Our high-level approach to reduce to a problem called Concatenated Element Distinctness, and then speed up that problem via packing into words. We assume without loss of generality that the graph is bipartite: by imposing a random bipartition and deleting edges that violate it, any 4-cycle is preserved with constant nonzero probability. We now show that it suffices to solve 4-cycle in graphs with maximum degree $O(\sqrt{n})$.

▶ **Lemma 25.** *Suppose a bipartite graph $G = (L \cup R, E)$ on $n > 2$ nodes is 4-cycle free. Then for all $d \geq \sqrt{2n}$, G has at most $2n/d$ nodes with degree at least d.*

Thus, the number of high-degree nodes is relatively small, which allows us to brute-force over them in subquadratic time. In particular, we can run the following preprocessing step.

▶ **Lemma 26.** *Given a bipartite graph $G = (V, E)$, there is a $O(n^{3/2})$ preprocessing algorithm that either returns "4-cycle" or produces a graph G' with maximum degree $\Delta = \sqrt{2n}$ such that solving 4-cycle on G and G' gives the same answer.*

Because of this, it suffices to solve 4-cycle in the following setting:

▶ **Definition 27** (4-Cycle Detection)**.** *Given a graph with $m = O(n^{3/2})$ edges and maximum degree $\Delta = O(\sqrt{n})$, decide whether there exists a 4-cycle.*

▶ **Definition 28** (Concatenated Element Distinctness). *Given k column vectors $L_1, \ldots, L_k$ of n ℓ-bit elements each, for every pair i, j of columns, element-wise concatenate them to a produce a new list $L_{i,j}$ containing n length-2ℓ elements. If for all i, j, the list $L_{i,j}$ has distinct elements, accept, else reject.*

We now reduce 4-cycle detection to this problem, and then use the word tricks from Section 3 on Concatenated Element Distinctness to improve the 4-cycle runtime when we have large word size.

▶ **Reminder of Theorem 6.** For $T(n, k, \ell) \geq n + k + \ell$, if Concatenated Element Distinctness can be solved in $O(T(n, k, \ell))$ time, then 4-cycle detection in n-node graphs can be done in $O(T(n, \sqrt{n}, (\log_2 n)/2))$ time. This reduction succeeds with constant probability.

▶ **Lemma 29.** *There is a deterministic algorithm for Concatenated Element Distinctness, on $O(\sqrt{n})$ lists each containing n elements of size $O(\log n)$ that runs in $O(n^2 \log^2 n/w)$ time.*

Subquadratic 4-Cycle Finding in Very Dense Graphs

Recall that by Theorem 10, a 4-cycle free graph has at most $\frac{n}{4}\left(1 + \sqrt{4n-3}\right) = O(n^{3/2})$ edges. Here, we give a sublinear time (in the number of edges) algorithm that finds a 4-cycle when the graph is noticeably denser, i.e., when the number of edges is $m \geq n^{3/2+\delta}$. Recall that a *k-core* of G is a maximal subgraph G' such that every vertex has degree at least k.

▶ **Lemma 30.** *For all sufficiently large n, every graph G with $m = n^{3/2+\delta}$ edges has a $(\sqrt{n}+1)$-core containing at least $\Omega(\sqrt{m})$ vertices.*

We will use this fact to provide a sublinear (in the size of the graph) time algorithm for finding a 4-cycle when the graph is extremely dense.

▶ **Reminder of Theorem 8.** There is a Las Vegas algorithm that, given a graph with n vertices and $n^{3/2+\delta}$ edges for $\delta \in (0, \frac{1}{2})$, returns a 4-cycle and runs in expected $O(n^{5/4-\delta/2})$ time.

7 Conclusion

In this paper, we have shown how a variety of ideas can be applied to achieve faster algorithms for some tricky triangle problems at the heart of fine-grained complexity. We conclude with some of the most significant questions left open by our work.

- We have shaved $\log(n)$ and word factors from the running times of sparse graph problems, assuming a bound on the maximum degree. Can we still improve the running times of these algorithms in the more general case of sparse graphs without a degree bound? For example, is there an algorithm for All-Edges Sparse Triangle running in $o(m^{4/3})$ time on m-edge graphs, assuming optimal matrix multiplication? Our approach requires at least two of the nodes in the triangle to have low degree, which is not enough to apply low-degree/high-degree tricks such as those in [6].
- Are there *deterministic* algorithms for these triangle problems that have similar running times? All of our algorithms rely on the use of randomness. Note that since all of our algorithms use $O(\log n)$-wise independent hash functions and linear hashing for small k, we only require at most $\mathsf{poly}(\log n)$ bits of randomness.

- The All-Edges Sparse Triangle problem cannot be solved much faster than the obvious running time, under the 3SUM and APSP hardness hypotheses. Is there a fine-grained reduction from All-Edges Sparse Triangle to Sparse Triangle Detection, so that detecting a triangle faster on sparse (or low-degree) graphs would yield a faster algorithm for the all-edges case? Such a reduction is well-known in the dense case [37].
- Can 4-cycle detection be solved in $O(n^2 \log n)/w$ time on the Word RAM for $w \gg \Omega(\log n)$? Is there any reason to believe that such an algorithm may not exist?

References

1. Amir Abboud, Karl Bringmann, and Nick Fischer. Stronger 3-sum lower bounds for approximate distance oracles via additive combinatorics. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, pages 391–404, 2023. doi:10.1145/3564246.3585240.
2. Amir Abboud, Seri Khoury, Oree Leibowitz, and Ron Safier. Listing 4-Cycles. In *43rd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2023)*, volume 284 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSTTCS.2023.25.
3. Miklós Ajtai, János Komlós, and Endre Szemerédi. Sorting in c log n parallel steps. *Comb.*, 3(1):1–19, 1983. doi:10.1007/BF02579338.
4. Susanne Albers and Torben Hagerup. Improved parallel integer sorting without concurrent writing. *Inf. Comput.*, 136(1):25–51, 1997. doi:10.1006/INCO.1997.2632.
5. Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995. doi:10.1145/210332.210337.
6. Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997. doi:10.1007/BF02523189.
7. Arne Andersson, Peter Bro Miltersen, and Mikkel Thorup. Fusion trees can be implemented with AC0 instructions only. *Theor. Comput. Sci.*, 215(1-2):337–344, 1999. doi:10.1016/S0304-3975(98)00172-8.
8. Ilya Baran, Erik D. Demaine, and Mihai Pătraşcu. Subquadratic algorithms for 3sum. *Algorithmica*, 50(4):584–596, 2008. doi:10.1007/S00453-007-9036-3.
9. Djamal Belazzougui, Gerth Stølting Brodal, and Jesper Sindahl Nielsen. Expected linear time sorting for word size ω $(\log^2 n \log\log n)$. In *Scandinavian Workshop on Algorithm Theory*, pages 26–37. Springer, 2014. doi:10.1007/978-3-319-08404-6_3.
10. Philip Bille, Anna Pagh, and Rasmus Pagh. Fast evaluation of union-intersection expressions. In *Algorithms and Computation, 18th International Symposium, ISAAC 2007, Sendai, Japan, December 17-19, 2007, Proceedings*, volume 4835 of *Lecture Notes in Computer Science*, pages 739–750. Springer, 2007. doi:10.1007/978-3-540-77120-3_64.
11. J.A Bondy and M Simonovits. Cycles of even length in graphs. *Journal of Combinatorial Theory, Series B*, 16(2):97–105, 1974. doi:10.1016/0095-8956(74)90052-5.
12. William G Brown. On graphs that do not contain a Thomsen graph. *Canadian Mathematical Bulletin*, 9(3):281–285, 1966.
13. Timothy M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM J. Comput.*, 39(5):2075–2089, 2010. doi:10.1137/08071990X.
14. Timothy M. Chan, Virginia Vassilevska Williams, and Yinzhan Xu. Hardness for triangle problems under even more believable hypotheses: reductions from real apsp, real 3sum, and OV. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 1501–1514. ACM, 2022. doi:10.1145/3519935.3520032.
15. Timothy M. Chan and Yinzhan Xu. Simpler reductions from exact triangle. In *2024 Symposium on Simplicity in Algorithms, SOSA 2024, Alexandria, VA, USA, January 8-10, 2024*, pages 28–38. SIAM, 2024. doi:10.1137/1.9781611977936.4.

16 Ashok K. Chandra, Larry Stockmeyer, and Uzi Vishkin. Constant depth reducibility. *SIAM Journal on Computing*, 13(2):423–439, 1984. doi:10.1137/0213028.

17 Hagai Cohen and Ely Porat. Fast set intersection and two-patterns matching. *Theor. Comput. Sci.*, 411(40-42):3795–3800, 2010. doi:10.1016/J.TCS.2010.06.002.

18 Anirban Dasgupta, Ravi Kumar, and D Sivakumar. Sparse and lopsided set disjointness via information theory. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 517–528. Springer, 2012. doi:10.1007/978-3-642-32512-0_44.

19 Bolin Ding and Arnd Christian König. Fast set intersection in memory. *Proc. VLDB Endow.*, 4(4):255–266, 2011. doi:10.14778/1938545.1938550.

20 David Eppstein, Michael T. Goodrich, Michael Mitzenmacher, and Manuel R. Torres. 2-3 cuckoo filters for faster triangle listing and set intersection. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 247–260. ACM, 2017. doi:10.1145/3034786.3056115.

21 P. Erdős, A. Rényi, and V. T. Sós. On a problem of graph theory. *Studia Sci. Math. Hungar.*, 1:215–235, 1966.

22 Paul Erdös and Miklós Simonovits. Supersaturated graphs and hypergraphs. *Comb.*, 3(2):181–192, 1983. doi:10.1007/BF02579292.

23 Alireza Farhadi, Mohammad Taghi Hajiaghayi, Kasper Green Larsen, and Elaine Shi. Lower bounds for external memory integer sorting via network coding. *SIAM J. Comput.*, 52(on):STOC19–87–STOC19–111, 2019. doi:10.1137/20M1321887.

24 Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.*, 5(1):83–89, 1976. doi:10.1137/0205006.

25 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, December 1993. doi:10.1016/0022-0000(93)90040-4.

26 Allan Grønlund and Seth Pettie. Threesomes, degenerates, and love triangles. *J. ACM*, 65(4):22:1–22:25, 2018. doi:10.1145/3185378.

27 Torben Hagerup. Simpler and faster dictionaries on the AC0 RAM. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 79–90. Springer, 1998. doi:10.1007/BFB0055042.

28 Ce Jin and Yinzhan Xu. Removing additive structure in 3sum-based reductions. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 405–418. ACM, 2023. doi:10.1145/3564246.3585157.

29 Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Dynamic set intersection. In *Algorithms and Data Structures - 14th International Symposium, WADS 2015, Victoria, BC, Canada, August 5-7, 2015. Proceedings*, volume 9214 of *Lecture Notes in Computer Science*, pages 470–481. Springer, 2015. doi:10.1007/978-3-319-21840-3_39.

30 Zongpeng Li and Baochun Li. Network coding: The case of multiple unicast sessions. *Proceedings of the 42nd Allerton Annual Conference on Communication, Control, and Computing*, October 2004.

31 Andrea Lincoln, Adam Polak, and Virginia Vassilevska Williams. Monochromatic triangles, intermediate matrix products, and convolutions. In *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, pages 53:1–53:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ITCS.2020.53.

32 Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 603–610. ACM, 2010. doi:10.1145/1806689.1806772.

33 Dana Richards and Arthur L Liestman. Finding cycles of a given length. In *North-Holland Mathematics Studies*, volume 115, pages 249–255. Elsevier, 1985.

34 Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM J. Discret. Math.*, 8(2):223–250, 1995. doi:10.1137/S089548019223872X.

35 Larry J. Stockmeyer and Uzi Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, 1984. doi:10.1137/0213027.

36 Virginia Vassilevska, Ryan Williams, and Raphael Yuster. Finding heaviest H-subgraphs in real weighted graphs, with applications. *ACM Trans. Algorithms*, 6(3):44:1–44:23, 2010. doi:10.1145/1798596.1798597.

37 Virginia Vassilevska Williams and R. Ryan Williams. Subcubic equivalences between path, matrix, and triangle problems. *J. ACM*, 65(5):27:1–27:38, 2018. doi:10.1145/3186893.

38 Virginia Vassilevska Williams and Ryan Williams. Finding, minimizing, and counting weighted subgraphs. *SIAM J. Comput.*, 42(3):831–854, 2013. doi:10.1137/09076619X.

39 Virginia Vassilevska Williams and Yinzhan Xu. Monochromatic triangles, triangle listing and APSP. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 786–797. IEEE, 2020. doi:10.1109/FOCS46700.2020.00078.

40 R. Ryan Williams. Faster all-pairs shortest paths via circuit complexity. *SIAM J. Comput.*, 47(5):1965–1985, 2018. doi:10.1137/15M1024524.