More Logarithmic-factor Speedups for 3SUM, (median,+)-convolution, and Some Geometric 3SUM-hard Problems

TIMOTHY M. CHAN, University of Illinois at Urbana-Champaign, USA

This article presents an algorithm that solves the 3SUM problem for n real numbers in $O((n^2/\log^2 n)(\log\log n)^{O(1)})$ time, improving previous solutions by about a logarithmic factor. Our framework for shaving off two logarithmic factors can be applied to other problems, such as (median,+)-convolution/matrix multiplication and algebraic generalizations of 3SUM. This work also obtains the first subquadratic results on some 3SUM-hard problems in computational geometry, for example, deciding whether (the interiors of) a constant number of simple polygons have a common intersection.

CCS Concepts: • **Theory of computation → Design and analysis of algorithms**; *Computational geometry*;

Additional Key Words and Phrases: 3SUM, convolution, matrix multiplication, computational geometry

ACM Reference format:
Timothy M. Chan. 2019. More Logarithmic-factor Speedups for 3SUM, (median,+)-convolution, and Some Geometric 3SUM-hard Problems. *ACM Trans. Algorithms* 16, 1, Article 7 (November 2019), 23 pages.
https://doi.org/10.1145/3363541

1 INTRODUCTION

3SUM. The starting point of this article is the *3SUM* problem:

> Given sets A, B, and C of n real numbers, decide whether there exists a triple $(a, b, c) \in A \times B \times C$ with $c = a + b$.[1]

The problem has received considerable attention by algorithm researchers, and understanding the complexity of the problem is fundamental to the field. The conjecture that it cannot be solved in $O(n^{2-\varepsilon})$ time (in the real or integer setting) has been used as a basis for proving conditional lower bounds for numerous problems from a variety of areas (computational geometry, data structures, string algorithms, and so on). See previous papers (such as Reference [22]) for more background.

[1]The original formulation seeks a triple with $a + b + c = 0$, which is equivalent after negating the elements of C.

A preliminary version of this article appeared in *Proceedings of the 29th ACM-SIAM Symposium on Discrete Algorithms (SODA'18)*, pp. 881–897.
Author's address: T. M. Chan, Department of Computer Science, University of Illinois at Urbana-Champaign, 201 North Goodwin Avenue, Urbana, IL, 61801, USA; email: tmc@illinois.edu.

1549-6325/2019/11-ART7 $15.00
https://doi.org/10.1145/3363541

In a surprising breakthrough, Grønlund and Pettie [22] discovered the first subquadratic algorithms for 3SUM. They showed the decision-tree complexity of the problem is $O(n^{3/2}\sqrt{\log n})$, and gave a randomized $O((n^2/\log n)(\log\log n)^2)$-time algorithm and a deterministic $O((n^2/\log^{2/3} n)(\log\log n)^{2/3})$-time algorithm in the standard real-RAM model. Small improvements were subsequently reported by Freund [19] and Gold and Sharir [21], who independently found $O((n^2/\log n)\log\log n)$-time deterministic algorithms (the latter also eliminated the $\sqrt{\log n}$ factor from the decision-tree complexity bound). In another dramatic breakthrough, Kane et al. [24] developed a technique for obtaining near-optimal decision-tree complexity for many problems, and in particular, a near-linear $O(n\log^2 n)$ decision-tree upper bound for 3SUM; their technique does not seem to have new implications on the (uniform) time complexity of 3SUM (although it probably could lead to yet another $O((n^2/\log n)(\log\log n)^{O(1)})$-time algorithm).

In this article, we give a further improvement on the time complexity, by about one more logarithmic factor: we present a new deterministic $O((n^2/\log^2 n)(\log\log n)^{O(1)})$-time algorithm.

Ignoring $\log\log n$ factors, this matches known results for 3SUM in the special case of integer input, where a randomized $O((n^2/\log^2 n)(\log\log n)^2)$ time bound can be obtained via hashing techniques [4]. In contrast, besides being more general, our algorithm is deterministic and can also solve variations of the problem, e.g., finding the closest number (predecessor or successor) in $\{a+b : (a,b)\in A\times B\}$ to each $c\in C$.

The development on 3SUM parallels the history of combinatorial algorithms for the *all-pairs shortest paths (APSP)* problem for dense real-edge-weighted graphs, or equivalently, the *(min,+)-matrix multiplication* problem for real matrices. In fact, the slightly subquadratic algorithms by Grønlund and Pettie and the subsequent refinements by Freund or Gold and Sharir all used a geometric subproblem, *dominance searching* in logarithmic dimensions, following the author's $O(n^3/\log n)$-time APSP algorithm [9]. A later paper by the author [10] gave a further improved $O((n^3/\log^2 n)(\log\log n)^{O(1)})$-time algorithm for APSP, using a different geometric approach, via *cuttings* in near-logarithmic dimensions. Our improvement will follow the approach in Reference [10]. The analogy between these 3SUM and APSP algorithms makes sense in hindsight, but is not immediately obvious, at least to this author (which explains why this article was not written right after Grønlund and Pettie's breakthrough), and there are some new technical challenges (for example, involving bit packing and, at some point, simulation of sorting networks).

(select,+)-Convolution/Matrix Multiplication. The way Grønlund and Pettie and subsequent authors used dominance to solve 3SUM actually more resembled a previous algorithm by Bremner et al. [6] on another related problem, *(median,+)-convolution*, or more generally, what we will call *(select,+)-convolution*:

> Given real sequences $A=\langle a_1,\ldots,a_n\rangle$ and $B=\langle b_1,\ldots,b_n\rangle$ and integers $k_1,\ldots,k_n$, compute c_i = the k_ith smallest of $\{a_u+b_{i-u} : u\in\{1,\ldots,i-1\}\}$, for every $i\in\{1,\ldots,n\}$.

(This problem was used to solve the "necklace alignment problem" under ℓ_1 distances [6].) Our ideas can also improve Bremner et al.'s $O((n^2/\log n)(\log\log n)^{O(1)})$ time bound to $O((n^2/\log^2 n)(\log\log n)^{O(1)})$ for that problem.

One can similarly solve the *(select,+)-matrix multiplication* problem in $O((n^3/\log^2 n)(\log\log n)^{O(1)})$ time (we are not aware of any prior work on this problem):

> Given two real $n\times n$ matrices $A=\{a_{ij}\}$ and $B=\{b_{ij}\}$ and an integer matrix $K=\{k_{ij}\}$, compute c_{ij} = the k_{ij}th smallest of $\{a_{iu}+b_{uj} : u\in\{1,\ldots,n\}\}$, for every $i,j\in\{1,\ldots,n\}$.

Algebraic 3SUM. Barba et al. [5] recently gave an $O((n^2/\sqrt{\log n})(\log\log n)^{O(1)})$-time algorithm for a generalization of 3SUM, which we will refer to as *algebraic 3SUM*:

> Let $\varphi : \mathbb{R}^2 \to \mathbb{R}$ be a function of *constant description complexity*, i.e., $\{(x, y, z) \in \mathbb{R}^3 : \varphi(x, y) = z\}$ is a semi-algebraic set of constant degree. Given sets A, B, and C of n real numbers, decide whether there exists a triple $(a, b, c) \in A \times B \times C$ with $\varphi(a, b) = c$.

Our ideas can also lead to a faster $O((n^2/\log^2 n)(\log\log n)^{O(1)})$-time algorithm for this problem.

Geometric 3SUM-Hard Problems. The importance of 3SUM stems from the many problems that were shown to be 3SUM-hard, starting with the seminal paper by Gajentaan and Overmars [20] in computational geometry. Grønlund and Pettie's breakthrough was exciting, because it gave hope to the possibility that slightly subquadratic algorithms might exist for these geometric problems as well. However, thus far, no such geometric result has materialized. (In contrast, examples of "*orthogonal-vectors*-hard" problems in computational geometry with slightly subquadratic algorithms were known earlier [1, 7, 8, 21].) Barba et al.'s work came close: their algebraic 3SUM algorithm can be used to solve a standard geometric 3SUM-hard problem—testing whether a given set of n points is degenerate, i.e., contain a collinear triple—but only in the special case when the points lie on a small number of fixed-degree algebraic curves in 2D. The input there is still intrinsically one-dimensional.

In this article, we show how to adapt our techniques to obtain an $O((n^2/\log^2 n)(\log\log n)^{O(1)})$-time algorithm for some genuinely two-dimensional problems:

(i) *Intersection of 3 polygons.* Given 3 simple polygons with n vertices, decide whether the common intersection is empty.[2]
(ii) *Coverage by 3 polygons.* Given 3 simple polygons with n vertices and a triangle Δ_0, decide whether the union of the polygons covers Δ_0.
(iii) *Degeneracy testing for $O(1)$-chromatic line segments.* Given $O(1)$ sets each consisting of n *disjoint* line segments in $\mathbb{R}^2$, decide whether there exist three line segments meeting at a common point.
(iv) *Offline triangle range searching for bichromatic segment intersections.* Given sets A and B each consisting of n *disjoint* line segments, and a set C of n triangles in $\mathbb{R}^2$, count the number of intersection points between A and B that are inside each triangle $c \in C$.[3]

Furthermore, we obtain a slightly slower $O((n^2/\log n)(\log\log n)^{O(1)})$-time algorithm for the following related problems:

(v) *Intersection of $O(1)$ polygons.* Given $O(1)$ simple polygons with n vertices, decide whether the common intersection is empty.
(vi) *Coverage by $O(1)$ polygons.* Given $O(1)$ simple polygons with n vertices and a triangle Δ_0, decide whether the union of the polygons covers Δ_0.
(vii) *Offline reverse triangle range searching for bichromatic segment intersections.* Given sets A and B each consisting of n *disjoint* line segments, and a set C of n triangles in $\mathbb{R}^2$, count the number of triangles in C containing each intersection point q between A and B. The output counts may be stored in some "implicit" representation—more precisely, in a data

[2] This problem was observed to be 3SUM-hard in an unpublished note http://tcs.postech.ac.kr/workshops/kwcg05/problems/smallest_circle.pdf (2005).
[3] An online query version of this problem was recently considered by de Berg et al. [15], who also briefly noted connection to 3SUM.

structure that can return the count for any given intersection point q in constant time, and that also stores explicitly the minimum or maximum count over all intersection points.

It is not difficult to see that all the above problems are 3SUM-hard. Moreover, (i)–(iii) reduce to (iv), and (v)–(vi) reduce to (vii): (i) reduces to (iv), since if the intersection of the three polygons is nonempty, one can triangulate the polygons, and some vertex or some intersection point between two edges of two polygons would be inside a triangle of the other polygon; (ii) reduces to (i) by taking complements; (iii) reduces to (iv) by examining each triple of sets and viewing line segments as degenerate triangles; (v) reduces to (vii), since if the intersection of the polygons is nonempty, one can triangulate the complements of polygons, and some vertex or some intersection between 2 edges of 2 polygons would lie in no triangles of the complements of the other polygons; (vi) reduces to (v) by taking complements. The results for (i), (ii), (v), and (vi) hold even if each polygon has holes (or is a disconnected collection of disjoint polygons). The time bound remains subquadratic for a nonconstant number of polygons, if the number is less than $\log^{\delta} n$ for some δ.

To summarize, our main contributions are:

- a general recipe on how to shave off (in most cases) two logarithmic factors for a host of problems—for "Four-Russians"-style algorithms, two logarithmic factors are usually the most one could eliminate (see Section 8 for exceptions, which tend to be more limited in their applicability).
- identification of the first natural 3SUM-hard problems in computational geometry with slightly subquadratic algorithms—this is perhaps the most important "qualitative" contribution of the article, to those who care less about the precise number of logarithmic factors shaved.

2 PRELIMINARIES

Let $[n]$ denote $\{1, 2, \ldots, n\}$. For a list A, let $A[i]$ denote its ith element.

We assume a real-RAM model, where a word can hold an input real number or a w-bit number/pointer for a fixed $w = \Omega(\log n)$. The only operations on reals are evaluating the signs of constant-degree polynomials over a constant number of input values (in fact, for our 3SUM algorithms, all comparisons on reals will be of the form $a + b \leq c$ or $a + b \leq a' + b'$). For convenience, we assume that the model supports some nonstandard operations on w-bit words in constant time. At the end, by setting $w = \delta_0 \log n$ for a sufficiently small constant $\delta_0 > 0$, nonstandard word operations with $O(1)$ arguments can be simulated by table lookup after an initial preprocessing of $2^{O(w)} = n^{O(\delta_0)}$ time.

We mention some simple facts about bit packing that will be of use later:

Fact 2.1. *Let $\langle x_1, \ldots, x_\ell \rangle$ be a sequence of ℓ numbers in $[b]$ stored in $O((\ell \log b)/w + 1)$ words.*

(a) One can sort the sequence in $O((\ell \log^2(b\ell))/w + 1)$ time.
(b) Given a table $f : [b] \to [b]$, one can compute $\langle f[x_1], \ldots, f[x_\ell] \rangle$ in $O((\ell \log^2(b\ell))/w + b)$ time.

Proof. (Review) Part (a) is well known, and is described, for example, in Reference [10, Lemma 2.3(a)]: Roughly, we use a packed variant of mergesort, with $O(\log \ell)$ rounds of merging, where each round takes time linear in the number of words, i.e., $O((\ell \log b)/w)$ (this may require some nonstandard word operations).

Part (b) can be found in Reference [10, Lemma 2.3(d)]: Roughly, we sort the list of pairs (i, x_i) by x_i using (a), split the list into sublists with a common x_i, replace x_i with $f[x_i]$ in each sublist, concatenate these sublists, and finally sort all pairs by i to get back the original order. □

3 3SUM

For two lists A and B, let $A + B$ denote the multiset $\{a + b : a \in A,\ b \in B\}$.

3.1 Reduction to Batched $A + B$ Searching/Selection

We formulate two subproblems, which will be the key to solving 3SUM:

Problem 3.1 (Batched $A + B$ Searching). *Given lists ("groups") $A_1, \ldots, A_m, B_1, \ldots, B_m$ of d real numbers each, and given "query" lists C_{ij} of values in $\mathbb{R}$ with $\sum_{i,j} |C_{ij}| = Q$, find the predecessor of c in $A_i + B_j$ for each $c \in C_{ij}$ and each $i, j \in [m]$.*

Problem 3.2 (Batched $A + B$ Selection). *Given lists ("groups") $A_1, \ldots, A_m, B_1, \ldots, B_m$ of d real numbers each, and given "query" lists K_{ij} of numbers in $[d^2]$ with $\sum_{i,j} |K_{ij}| = Q$, find the kth smallest in $A_i + B_j$ for each $k \in K_{ij}$ and each $i, j \in [m]$.*

Let $T_{\text{search},+}(m, d, Q)$ and $T_{\text{select},+}(m, d, Q)$ be the time complexity of these two problems. Note that $T_{\text{search},+}(m, d, Q) = O((T_{\text{select},+}(m, d, Q) + Q + m^2) \cdot \log d)$, since Problem 3.1 reduces to $O(\log(d^2))$ instances of Problem 3.2, by simultaneous binary searches for the rank of c for all $c \in C_{ij}$ and all i, j.

3SUM clearly reduces to Problem 3.1 with $m = 1$ and $d = Q = n$. The following simple lemma, based on Grønlund and Pettie's grouping approach [22], shows a better reduction, to instances with smaller group size d:

Lemma 3.3. *3SUM can be solved in time $O(T_{\text{search},+}(n/d, d, n^2/d) + n^2/d + n \log n)$ for any given $d \le n$.*

Proof. Sort A and B. Divide A into sublists $A_1, \ldots, A_{n/d}$ of size d, and B into sublists $B_1, \ldots, B_{n/d}$ of size d. For each $c \in C$, put c in C_{ij} iff $c \in [A_i[1] + B_j[1], A_{i+1}[1] + B_{j+1}[1])$. For each c, the (i, j) pairs satisfying this condition form a monotone sequence in $[n/d]^2$ (nondecreasing in the first coordinate and nonincreasing in the second); thus, there are $O(n/d)$ such pairs and they can be found by a linear scan over the two lists $\langle A_1[1], \ldots, A_{n/d}[1]\rangle$ and $\langle B_1[1], \ldots, B_{n/d}[1]\rangle$ in $O(n/d)$ time. So, $\sum_{i,j} |C_{ij}| = O(n^2/d)$, and the total time to generate all C_{ij}'s is $O(n^2/d)$. For each $c \in C$, the predecessor of c in $A + B$ is the maximum among the predecessors of c in $A_i + B_j$ for all (i, j) with $c \in C_{ij}$. This gives an instance of Problem 3.1 with $O(n/d)$ sets of size d and total query list size $Q = O(n^2/d)$. □

3.2 Batched $A + B$ Selection via Cuttings in Near-logarithmic Dimensions

We now solve the batched $A + B$ selection problem for group size d near $\log m$, using a geometric approach based on the author's APSP algorithm [10]. One needs one tool from computational geometry:

Fact 3.4 (Cutting Lemma). *Given $r \le n$ and a set H of n hyperplanes in $\mathbb{R}^d$, we can cut $\mathbb{R}^d$ into $d^{O(d)} r^{O(d)}$ disjoint cells so that each cell is crossed[4] by at most n/r hyperplanes of H.*

Furthermore, given a set P of n points, for every cell Δ with $P \cap \Delta \ne \emptyset$, one can generate the set $P_\Delta = P \cap \Delta$ and a set H_Δ of size at most n/r containing all hyperplanes of H crossing Δ, in $d^{O(d)} n r^{O(d)}$ total time.

Proof (Review). If randomization is allowed, then a simple algorithm is to draw a random sample of about r hyperplanes and take the cells from a *canonical triangulation* of the *arrangement* of these hyperplanes. Derandomization is more involved. See Reference [14] for a survey.

[4] One says that h crosses Δ if h intersects the interior of Δ (or the relative interior of Δ, in the case that Δ is not full-dimensional).

In our application, the hyperplanes are orthogonal to only a small number ($d^{O(1)}$) of directions. In this case, there is a much easier proof, as noted in Reference [10]: Roughly, select r hyperplanes (r quantiles) for each direction, and take the cells from the arrangement of the resulting $d^{O(1)}r$ hyperplanes. □

We begin with one solution to batched $A + B$ selection in Theorem 3.5 below. It will be superseded by the later version of Theorem 3.6 but serves as a good warm-up of the basic idea.

Theorem 3.5 (Warm-up Version). *$T_{\text{select},+}(m, d, Q) = O(Q + m^2)$ for $d \le \delta \log m / \log\log m$ for a sufficiently small constant $\delta > 0$.*

Proof. For each A_i, define a point $p_i = (A_i[1], \ldots, A_i[d]) \in \mathbb{R}^d$. For each B_j, define a set of $O(d^4)$ hyperplanes

$$H_j = \left\{ \{(x_1, \ldots, x_d) \in \mathbb{R}^d : x_u + B_j[v] = x_{u'} + B_j[v']\} \quad : \quad u, v, u', v' \in [d] \right\}.$$

Let P be the resulting set of m points and H be the resulting set of $O(d^4 m)$ hyperplanes. Apply the Cutting Lemma to H and P, and obtain sets H_Δ and P_Δ in time $d^{O(d)} m r^{O(d)}$, where each H_Δ has size at most $O(d^4 m/r)$ and the number of cells Δ is $d^{O(d)} r^{O(d)}$. Take a cell Δ and an index $j \in [m]$. We describe how to handle the queries for K_{ij} for all $p_i \in P_\Delta$.

- Case 1: some hyperplane of H_j is in H_Δ. Since $|H_\Delta|$ is small, there are not too many such j's and one can afford to use a slow algorithm here:
  Namely, for each $p_i \in P_\Delta$, just sort $A_i + B_j$. Then, one can look up the answer for each $k \in K_{ij}$ in constant time. The total time is $O(\sum_{p_i \in P_\Delta} |K_{ij}| + |P_\Delta| \cdot d^2 \log d)$.
  Since there are at most $|H_\Delta| = O(d^4 m/r)$ choices of j for each Δ and $\sum_\Delta |P_\Delta| = m$, the total time over all Δ and all j is $O(Q + (d^4 m/r) \cdot m \cdot d^2 \log d)$.
- Case 2: no hyperplane of H_j is in H_Δ. One can save time here by observing that the sorted ordering of $A_i + B_j$ is the same as the sorted ordering of $A_{i'} + B_j$ for every $p_i, p_{i'} \in P_\Delta$, because no hyperplane of H_j crosses Δ. In particular, the index pair for the kth smallest in $A_i + B_j$ is the same as the index pair for the kth smallest in $A_{i'} + B_j$ for every $p_i, p_{i'} \in P_\Delta$.
  So, pick one representative point p_{i_Δ} in P_Δ, and just sort $A_{i_\Delta} + B_j$. Then, one can look up the answer for each $k \in K_{ij}$ in constant time. The total time is $O(\sum_{p_i \in P_\Delta} |K_{ij}| + |P_\Delta| + d^2 \log d)$.
  Since there are $d^{O(d)} r^{O(d)}$ cells Δ and $\sum_\Delta |P_\Delta| = m$, the total time over all Δ and all $j \in [m]$ is $O(Q + m^2 + (dr)^{O(d)} \cdot m \cdot d^2 \log d)$.

The overall running time is $O(Q + m^2 + d^6 (m^2/r) \log d + (dr)^{O(d)} m)$. Setting $r := d^7$ gives the desired result for $d \le \delta \log m / \log\log m$. □

It follows that $T_{\text{search},+}(m, Q, d) = O((Q + m^2) \log d)$ for $d = \delta \log m / \log\log m$, and so by Lemma 3.3, we immediately obtain a 3SUM algorithm with running time $O((n^2/d) \log d) = O((n^2/\log n)(\log\log n)^2)$. (Note that our solution here is simpler than previous solutions of similar running time [19, 21].)

To improve Theorem 3.5 by about a w factor, we use bit-packing techniques. (Note that such an improvement does not seem to work for the previous solutions [19, 21] that were based on dominance instead of cuttings.) In Problem 3.2, note that each query list K_{ij} can be stored compactly in $O((|K_{ij}| \log d)/w + 1)$ words. So can the output for K_{ij}: if the kth smallest of $A_i + B_j$ is $A_i[u] + B_j[v]$, it can be represented as an index pair $(u, v) \in [d]^2$. Thus, the total input/output size is $O((Q \log d)/w + m^2)$ in words.

We will actually use a variant of the input/output representation that lowers the m^2 term: Divide $[m]$ into $m/\overline{w}$ blocks of $\overline{w}$ consecutive indices for a fixed value $\overline{w} \ge w$. For each $i \in [m]$ and block

β, we store a list $K_{i\beta}$ containing $(j \bmod \overline{w}, k)$ over all $j \in \beta$ and $k \in K_{ij}$. The input/output size for this $\overline{w}$-*block representation* becomes $O((Q\log(d\overline{w}))/w + m^2/\overline{w})$ in words.

Theorem 3.6 (Refined Version). *$T_{\text{select},+}(m,d,Q) = O((Q\log^2(d\overline{w}))/w + m^2/\overline{w})$ for $d \le \delta \log m/\log(\overline{w}\log m)$ for a sufficiently small constant $\delta > 0$.*

Proof. We modify the proof of Theorem 3.5. Define H, P, and the cutting as before.

Take a cell Δ and a block β of $\overline{w}$ consecutive indices in $[m]$. We describe how to handle the queries for $K_{i\beta}$ for all $p_i \in P_\Delta$.

- Case 1: some hyperplane of $\bigcup_{j\in\beta} H_j$ is in H_Δ.
  Take a fixed $p_i \in P_\Delta$. Sort $A_i + B_j$ for all $j \in \beta$ in $O(\overline{w}d^2\log d)$ time. Create a table $f : \{0,\ldots,\overline{w}-1\}\times[d^2] \to [d]^2$ where $f[j \bmod \overline{w}, k]$ stores the index pair for the kth smallest of $A_i + B_j$. Look up f for the answers for $K_{i\beta}$ in $O((|K_{i\beta}|\log^2(d\overline{w}))/w + \overline{w}d^2)$ time by Fact 2.1(b). The total time over all $p_i \in P_\Delta$ is $O(\sum_{i\in P_\Delta}|K_{i\beta}|\log^2(d\overline{w}))/w + |P_\Delta|\cdot\overline{w}d^2\log d)$.
  Since there are at most $|H_\Delta| = O(d^4m/r)$ choices of β for each Δ and $\sum_\Delta |P_\Delta| = m$, the total time over all Δ and all β is $O((Q\log^2(d\overline{w}))/w + (d^4m/r)\cdot m\cdot\overline{w}d^2\log d)$.
- Case 2: no hyperplane of $\bigcup_{j\in\beta} H_j$ is in H_Δ. Observe that the sorted ordering of $A_i + B_j$ is the same as the sorted ordering of $A_{i'} + B_j$ for every $p_i, p_{i'} \in P_\Delta$ and every $j \in \beta$.
  So, pick one representative point p_{i_Δ} in P_Δ. Sort $A_{i_\Delta} + B_j$ for all $j \in \beta$ in $O(\overline{w}d^2\log d)$ time, and create a table $f : \{0,\ldots,\overline{w}-1\}\times[d^2]\to[d]^2$ where $f[j \bmod \overline{w}, k]$ stores the index pair for the kth smallest of $A_{i_\Delta} + B_j$. Concatenate the lists $K_{i\beta}$ over all $p_i \in P_\Delta$, look up f for the answers for the combined list in $O(\sum_{p_i\in P_\Delta}(|K_{i\beta}|\log^2(d\overline{w}))/w + \overline{w}d^2 + |P_\Delta|)$ time by Fact 2.1(b), then split the output list to get the answers for each $K_{i\beta}$.
  Since there are $d^{O(d)}r^{O(d)}$ cells Δ and $\sum_\Delta |P_\Delta| = m$, the total time over all Δ and $m/\overline{w}$ blocks β is $O((Q\log^2(d\overline{w}))/w + (dr)^{O(d)}\cdot(m/\overline{w})\cdot\overline{w}d^2\log d + (m/\overline{w})\cdot m)$.

The overall running time is at most $O((Q\log^2(d\overline{w}))/w + m^2/\overline{w} + d^6\overline{w}(m^2/r)\log d + (dr)^{O(d)}m)$. Setting $r := d^7\overline{w}^2$ gives the desired result for $d \le \delta\log m/\log(\overline{w}\log m)$. □

Unfortunately, the above theorem in itself is not sufficient to yield a further improvement to 3SUM, because of the $O(n^2/d)$ term in Lemma 3.3, unless one could choose a bigger d. We will propose more sophisticated bit-packing tricks to get around this bottleneck.

3.3 Batched $A + B$ Comparisons/Sorting via Bit Packing and Fredman's Trick

First, one needs a subroutine for solving the following subproblem:

Problem 3.7 (Batched $A + B$ Comparisons). *Given lists ("groups") $A_1,\ldots,A_m,B_1,\ldots,B_m$ of d real numbers each, and given "query" lists Q_{ij} of quadruples in $[d]^4$ with $\sum_{i,j}|Q_{ij}| = Q$, test whether $A_i[u] + B_j[v] \le A_i[u'] + B_j[v']$ for each $(u,v,u',v') \in Q_{ij}$ and each $i,j \in [m]$.*

Let $T_{\text{comp},+}(m,d,Q)$ be the time complexity of this problem. Note that by bit packing, the total input size is $O((Q\log d)/w + m^2)$ in words, and the total output size is $O(Q/w + m^2)$ in words, since the output to each Q_{ij} is a $|Q_{ij}|$-bit vector. (One will not need block representations here.) The following theorem is inspired by Reference [10, Theorem 2.4] (which in turn was based on some ideas from an APSP algorithm of Han [23]):

Theorem 3.8. *$T_{\text{comp},+}(m,d,Q) = O((Q\log^2 d)/w + m^2)$ for $m \ge d^2\log d$.*

Proof. We use "Fredman's trick," i.e., the (trivial) observation that

$$A_i[u] + B_j[v] \le A_i[u'] + B_j[v'] \iff A_i[u] - A_i[u'] \le B_j[v'] - B_j[v].$$

(This observation was the key behind Fredman's first subcubic decision-tree result on APSP and (min,+)-matrix multiplication [18].)

Sort the $O(d^2m)$ elements

$$\{A_i[u] - A_i[u'] : i \in [m], u, u' \in [d]\} \cup \left\{B_j[v'] - B_j[v] : j \in [m], v, v' \in [d]\right\}$$

in $O(d^2m\log(dm))$ time. Let L be the resulting list. Create tables $f, g : [m] \times [d]^2 \to [d^2m]$ where $f[i,u,u']$ stores the rank of $A_i[u] - A_i[u']$ in L and $g[j,v',v]$ stores the rank of $B_j[v'] - B_j[v]$ in L.

Map each quadruple (u,u',v,v') in the list Q_{ij} to (i,j,u,u',v,v'). Concatenate the lists over all $(i,j) \in [m]^2$ in lexicographical order. In the concatenated list, map (i,j,u,u',v,v') to (i,u,u'), then to $f[i,u,u']$ by Fact 2.1(b); similarly, map (i,j,u,u',v,v') to (j,v,v'), then to $g[j,v,v']$ by Fact 2.1(b) again; all this takes $O((Q\log^2(dm))/w + m^2 + d^2m)$ time. Combine the two resulting lists to get a list of pairs $(f[i,u,u'], g[j,v,v'])$. One can then obtain the output list of Boolean values corresponding to whether $f[i,u,u'] \le g[j,v',v]$, and split the list by (i,j) to get the answers to each Q_{ij}. The total time is $O((Q\log^2(dm))/w + m^2 + d^2m\log(dm)) = O((Q\log^2(dm))/w + m^2)$ for $m \ge d^2\log d$.

The $\log(dm)$ factors in the above time bound can be replaced by $\log d$ easily: Let $m_0 := d^2\log d$. Divide $A_1, \ldots, A_m, B_1, \ldots, B_m$ into $O(m/m_0)$ blocks of $O(m_0)$ lists each. Apply the above algorithm to each pair of blocks. The total time is now $O((Q\log^2(dm_0))/w + (m/m_0)^2 \cdot m_0^2) = O((Q\log^2 d)/w + m^2)$. □

The above subroutine enables us to solve the following, tougher subproblem:

Problem 3.9 (Batched $A + B$ Sorting). *Given lists ("groups") $A_1, \ldots, A_m, B_1, \ldots, B_m$ of d real numbers each, and given "query" lists Q_{ij} with $\sum_{i,j} |Q_{ij}| = Q$, where each element of Q_{ij} is a sequence of ℓ pairs in $[d]^2$, reorder each sequence $\langle (u_1,v_1), \ldots, (u_\ell,v_\ell)\rangle$ in Q_{ij} so that at the end, $A_i[u_1] + B_j[v_1] \le \cdots \le A_i[u_\ell] + B_j[v_\ell]$, for each $i,j \in [m]$.*

Let $T_{\mathrm{sort},+}(m,d,\ell,Q)$ be the time complexity of this problem. Note that if $\ell \le \delta w/\log d$ for a sufficiently small constant δ, then each sequence in Q_{ij} can be packed in a word, and the total input/output size (in words) is $O(Q + m^2)$.

Theorem 3.10. *$T_{\mathrm{sort},+}(m,d,\ell,Q) = O((Q + m^2)\cdot\log^{O(1)}(dw))$ if $m \ge d^2\log d$ and $\ell \le \delta w/\log(dw)$ for a sufficiently small constant $\delta > 0$.*

Proof. The main idea is to simulate a *sorting network* to sort the sequences in Q_{ij} for all i,j simultaneously. Recall that a sorting network sorts ℓ elements $x_1, \ldots, x_\ell$ in rounds. In each round, we have a pre-chosen set of $O(\ell)$ disjoint pairs of indices, and we perform a *compare-and-exchange* operation on each such pair. A compare-and-exchange operation on an index pair $(r,r') \in [\ell]^2$ involves testing whether $x_r > x_{r'}$, and if true, swapping x_r and $x_{r'}$. (The pre-chosen set of index pairs per round is independent of the input values.) The AKS sorting network [3] can sort ℓ elements in $O(\log\ell)$ rounds.

Consider one round of the network. For each i,j and each sequence in Q_{ij}, we first extract the list of $O(\ell)$ pairs of elements that need to be compare in $O(1)$ time by using a (nonstandard) word operation, since the sequence is packed in a word, and the pre-chosen set of index pairs (r,r') can also be encoded in a word (as $\ell\log\ell \le \delta w$). The total time so far is $O(Q + m^2)$. We concatenate these lists over all sequences in Q_{ij} for each i,j, apply the algorithm for batched $A + B$ comparisons with total query list size $O(\ell Q)$, and split the answer list per i,j and per sequence. For each i,j and each sequence in Q_{ij}, one can then perform the necessary swaps in $O(1)$ time by another word operation, since each sequence is packed in a word and the outcomes of the comparisons fit in a word. Each round requires total time $O(T_{\text{comp},+}(m,d,\ell Q) + Q + m^2) = O((\ell Q \log^2 d)/w + Q + m^2) = O(Q \log d + m^2)$. The overall running time is multiplied by an $O(\log \ell)$ factor, by using the AKS sorting network. $\square$

Note that the above algorithm can sort Q_{ij} even when extra fields are attached to each pair in Q_{ij}, provided that the extra fields require $O(\log(dw))$ bits.

3.4 Putting Everything Together via 2-Level Grouping

We now solve the batched $A+B$ searching problem for a bigger group size d near $w \log m$, by reducing to batched $A+B$ selection for a smaller group size:

Lemma 3.11. *$T_{\text{search},+}(m,d,Q) = O((T_{\text{select},+}(\ell m, d/\ell, \ell Q) + T_{\text{sort},+}(m,d,\ell,Q) + \ell Q/w + Q + m^2 + \ell^2 m^2/\overline{w}) \cdot \log^{O(1)}(d\overline{w}) + dm \log d)$ for any given $\ell \le \min\{d, \overline{w}\}$.*

Proof. One may assume that ℓ and d are powers of 2. If $|C_{ij}| \ge d^2$, then one can afford to use a slow algorithm: compute $A_i + B_j$ in $O(d^2 \log d)$ time and test for each $c \in C_{ij}$ whether $c \in A_i + B_j$ by binary search. The number of such C_{ij}'s is at most $O(Q/d^2)$ and so the total time for this step is $O((Q/d^2) \cdot d^2 \log d + Q \log d) = O(Q \log d)$. Thus, one may assume that $|C_{ij}| < d^2$ from now on.

Sort each A_i and B_j. Divide each A_i into sublists $A_{i1}, \ldots, A_{i\ell}$ of size $d_0 := d/\ell$, and each B_j into sublists $B_{j1}, \ldots, B_{j\ell}$ of size d_0.

Consider a fixed $c \in C_{ij}$. We describe an algorithm search(c, A_i, B_j) to find the predecessor of c in $A_i + B_j$. Observe that if the answer is in $A_{ip} + B_{jq}$, then we either have (i) $c \in [A_{ip}[1] + B_{jq}[1], A_{ip}[1] + B_{j,q+1}[1])$, or (ii) $c \in [A_{ip}[1] + B_{j,q+1}[1], A_{i,p+1}[1] + B_{j,q+1}[1])$. One assumes the first case; the second case can be handled by a symmetric algorithm. The algorithm is given by the pseudocode below, and works as follows: For each $p \in [\ell]$, first find the unique index q_p for which $c \in [A_{ip}[1] + B_{jq_p}[1], A_{ip}[1] + B_{j,q_p+1}[1])$, by binary search (lines 2 and 3). Then find the predecessor of c in $A_{ip} + B_{jq_p}$ by another binary search, this time, over the ranks (lines 4–6), using an oracle for selection in $A_{ip} + B_{jq_p}$. The largest of the predecessors found gives us the overall predecessor of c in $A_i + B_j$.

search(c, A_i, B_j):
1. initialize $q_1 = \cdots = q_\ell = k_1 = \cdots = k_\ell = 1$
2. for $s = \ell/2, \ell/4, \ldots, 1$:
3.   for each $p \in [\ell]$, if $c > A_{ip}[1] + B_{j,q_p+s}[1]$, then $q_p := q_p + s$
4. for $s = d_0^2/2, d_0^2/4, \ldots, 1$:
5.   for each $p \in [\ell]$, if $c >$ (the (k_p+s)th smallest in $A_{ip} + B_{jq_p}$), then $k_p := k_p + s$
6. return max{the k_pth smallest in $A_{ip} + B_{jq_p} : p \in [\ell]$}

The main idea is to run algorithm search(c, A_i, B_j) simultaneously for all $c \in C_{ij}$ and all $i, j \in [m]$, using bit-packed lists. For each i, j, c, one maintains a list L_{ijc} of $O(\ell)$ tuples (p, q_p, k_p) for all $p \in [\ell]$ and variables q_p and k_p kept by the algorithm. There are $\sum_{i,j} |C_{ij}| = Q$ such lists L_{ijc}, which in total contain $O(\ell Q)$ tuples and require $O((\ell Q \log d)/w + Q)$ space.

To implement line 3, we first sort the tuples (p, q_p, k_p) in L_{ijc} by $A_{ip}[1] + B_{j,q_p+s}[1]$; by the batched $A+B$ sorting subroutine, this takes $O(T_{\text{sort},+}(m,d,\ell,Q))$ total time over all i, j, c. After sorting, one can resolve all comparisons between c and $A_{ip}[1] + B_{j,q_p+s}[1]$ for all $p \in [\ell]$, by standard binary search for c in $O(\log \ell)$ time per i, j, c (no bit packing needed here). The total time for all these binary searches is $O(Q \log d)$.

Line 5 (or 6) is similar. However, before one can sort, one needs to find the index pair for the $(k_p + s)$th smallest in $A_{ip} + B_{jq_p}$ for all $p \in [\ell]$. This can be done by the batched $A+B$ selection algorithm in $O(T_{\text{select},+}(\ell m, d_0, \ell Q))$ total time over all i, j, c. Before the call to batched $A + B$ selection, some setup is required to reformat to a $\overline{w}$-block input representation. More precisely, divide $[\ell m]$ into blocks of $\overline{w}$ indices. Instead of putting $k_p + s$ in a list $K_{i\ell+p, j\ell+q_p}$, one wants to put $((j \bmod (\overline{w}/\ell))\ell + q_p, k_p + s)$ in a list $K_{i\ell+p,\beta}$ where β is the block containing $j\ell$. To this end, first map each tuple (p, q_p, k_p) in L_{ijc} to $(p, q_p, k_p, j \bmod (\overline{w}/\ell), c)$ (note that c can be encoded as an integer in $[|C_{ij}|] = [O(d^2)]$); for each i and each block β, let $L_{i\beta}$ be the concatenation of L_{ijc} over all c and all j such that $j\ell \in \beta$; sort $L_{i\beta}$ by p (by Fact 2.1(a)), and split it into sublists $L_{ip\beta}$ with a common p; from $L_{ip\beta}$, we can then obtain $K_{i\ell+p,\beta}$ and are ready for the call. After the call, one can concatenate the answers for $L_{ip\beta}$ over all p to get the answers for $L_{i\beta}$, then sort $L_{i\beta}$ by $(j \bmod (\overline{w}/\ell), c)$ to get back the answers for L_{ijc}. Since there are $O(\ell m/\overline{w})$ choices for β and so $O(m \cdot \ell \cdot \ell m/\overline{w})$ choices for (i, p, β), these extra steps take $O((\ell Q \log^2(d\overline{w}))/w + \ell^2 m^2/\overline{w})$ total time.

Since the entire algorithm has $\log \ell + \log(d_0^2)$ iterations, the overall running time is multiplied by an $O(\log d)$ factor. □

Combining Lemma 3.11 and Theorems 3.6 and 3.10, one obtains

$$\begin{aligned} T_{\text{search},+}(m, d, Q) &= O((\ell Q/w + Q + m^2 + \ell^2 m^2/\overline{w}) \cdot \log^{O(1)}(d\overline{w})) \\ &= O((Q + m^2) \cdot \log^{O(1)} w) \end{aligned}$$

by setting $\ell \approx \delta w/\log(dw)$ and $\overline{w} := w^2$, assuming that $d \leq \delta^2 w \log m/\log^2 w$, and $w = \Omega(\log m)$, and $w \leq m^{o(1)}$.

By Lemma 3.3 with $d \approx \delta^2 w \log n/\log^2 w$, we obtain our main result:

Corollary 3.12. *3SUM can be solved in* $O((n^2/(w \log n)) \log^{O(1)} w) \leq O((n^2/\log^2 n)(\log\log n)^{O(1)})$ *time, assuming that* $w = \Omega(\log n)$ *and* $w \leq n^{o(1)}$.

We do not feel it is important to optimize the log log factors, but for those interested, the above 3SUM algorithm has 5 $\log\log n$ factors. If one does not care about $\log\log n$ factors, then one can replace the AKS sorting network by one of Batcher's sorting networks, which is simpler. With randomization, sorting networks can probably be avoided entirely (by using approximate medians instead of sorting).

4 (SELECT,+)-CONVOLUTION/MATRIX MULTIPLICATION

One can apply similar ideas to the (select,+)-matrix multiplication and (select,+)-convolution problems. For two lists A and B of d elements, let $A \hat{+} B$ denote the list $\langle A[1] + B[1], A[2] + B[2], \ldots, A[d] + B[d]\rangle$. Let $T_{\text{select},\hat{+}}(m, d, Q)$ be the time complexity of batched $A \hat{+} B$ selection, i.e., the variant of Problem 3.2 with + replaced by $\hat{+}$.

4.1 Reduction to Batched $A \hat{+} B$ Selection

We first show how (select,+)-matrix multiplication and (select,+)-convolution can be reduced to batched $A \hat{+} B$ selection for small group size d.

Lemma 4.1.

(a) *(select,+)-matrix multiplication can be solved in* $O((T_{\text{select},\hat{+}}(n, d, n^2) + n^2) \cdot (n/d) \log d)$ *time.*

(b) *(select,+)-convolution can be solved in* $O((T_{\text{select},\hat{+}}(n/d, d, n^2/d^2) + n^2/d^2) \cdot d \log d)$ *time.*

Proof. Given ℓ sorted arrays $X_1, \ldots, X_\ell$ of d elements each, there are known algorithms that can find the kth smallest in $O(\ell \log d)$ time [16, 17, 25]. Most such algorithms proceed in $O(\log d)$ iterations, where each iteration takes $O(\ell)$ time and requires looking up $O(1)$ entries per array in parallel. The exact details are not important for our purposes, but for the sake of completeness, we provide one such algorithm:

select$(k, X_1, \ldots, X_\ell)$:
1. initialize $k_1 = \cdots = k_\ell = 0$ and $s_1 = \cdots = s_p = d$
2. while $S := s_1 + \cdots + s_\ell > 20\ell$ do:
3. if $k \leq S/2$:
4. for each $p \in [\ell]$, let $\mu_p := X_p[k_p + \lfloor s_p/3 \rfloor]$ and $\nu_p := X_p[k_p + 2\lfloor s_p/3 \rfloor]$
5. let x be the smallest such that $\sum_{\mu_p \leq x} \lfloor s_p/3 \rfloor + \sum_{\nu_p \leq x} \lfloor s_p/3 \rfloor > k$
6. for each $p \in [\ell]$, if $\mu_p \geq x$, then $s_p := \lfloor s_p/3 \rfloor$, else if $\nu_p \geq x$, then $s_p := 2\lfloor s_p/3 \rfloor$
7. else negate and reverse $X_p[k_p + 1, \ldots, k_p + s_p]$ for each $p \in [\ell]$, and set $k := S - k + 1$
8. return the k-th smallest in $\bigcup_{p \in [\ell]} X_p[k_p + 1, \ldots, k_p + s_p]$

The above algorithm maintains the invariant that the answer is the kth smallest in $\bigcup_{p \in [\ell]} X_p[k_p + 1, \ldots, k_p + s_p]$. Consider the case when $k \leq S/2$. In line 5, the answer is at most x, because there are more than k elements at most x. Thus, in line 6, if $\mu_p \geq x$, then one can remove the top two-thirds of $X_p[k_p + 1, \ldots, k_p + s_p]$, else if $\nu_p \geq x$, then one can remove the top third. The number of remaining elements is at most $\sum_{\mu_p < x} \lfloor s_p/3 \rfloor + \sum_{\nu_p < x} \lfloor s_p/3 \rfloor + \sum_{p=1}^{\ell} (s_p - 2\lfloor s_p/3 \rfloor) \leq k + S/3 + 2\ell \leq 5S/6 + 2\ell$. The case when $k \geq S/2$ is symmetric (note that the negate-and-reverse operation can be done implicitly). It takes at most $O(\log d)$ iterations for S to drop from $d\ell$ to below 20ℓ. Lines 5 and 8 require $O(\ell)$ time by using a linear-time (weighted) selection algorithm, so the total time is indeed $O(\ell \log d)$.

Now, we are ready to prove the lemma:

(a) Let $\ell := n/d$. Divide the ith row of A into sublists $A_i^{(1)}, \ldots, A_i^{(\ell)}$ of d elements each, and divide the jth column of B into sublists $B_j^{(1)}, \ldots, B_j^{(\ell)}$ of d elements each. For each $i, j \in [n]$, one wants to compute c_{ij} = the k_{ij}th smallest of the union of $A_i^{(1)} \hat{+} B_j^{(1)}, \ldots, A_i^{(\ell)} \hat{+} B_j^{(\ell)}$. Let $X_{ij}^{(p)}$ denote the list $A_i^{(p)} \hat{+} B_j^{(p)}$ rearranged in sorted order. The idea is to run the above algorithm select$(k_{ij}, X_{ij}^{(1)}, \ldots, X_{ij}^{(\ell)})$ simultaneously for all $i, j \in [n]$. We do not explicitly store the sorted list $X_{ij}^{(p)}$, but when desired, can retrieve the kth element in the sorted list for any given k by selection in $A_i^{(p)} \hat{+} B_j^{(p)}$. Thus, in each iteration, line 4 can be done by ℓ instances of batched $A \hat{+} B$ selection (one instance per p); the total time over all i, j is $O((T_{\text{select},\hat{+}}(n, d, n^2) + n^2) \cdot \ell)$. Line 5 takes $O(\ell)$ time per i, j, for a total of $O(n^2\ell)$ time. Since the algorithm has $O(\log d)$ iterations, the overall running time is multiplied by an $O(\log d)$ factor.

(b) Divide A into $\ell := n/d$ sublists $A_1, \ldots, A_\ell$ of size d each, and B into sublists $B_1, \ldots, B_\ell$ of size d each. For each $i \in [\ell]$, one wants to compute c_{id+1} = the k_{id+1}th smallest of the union of $A_1 \hat{+} B_i^r, A_2 \hat{+} B_{i-1}^r, \ldots, A_i \hat{+} B_1^r$, where L^r denotes the reverse of L. Let X_{ip} denote the list $A_p \hat{+} B_{i-p+1}^r$ rearranged in sorted order. The idea is to run the algorithm select$(k_i, X_{i1}, \ldots, X_{ii})$ simultaneously for all $i \in [\ell]$. In each iteration, line 4 can be done by batched $A \hat{+} B$ selection; the total time over all i is $O(T_{\text{select},\hat{+}}(\ell, d, \ell^2) + \ell^2)$. Line 5 takes $O(\ell)$ time per i, for a total of $O(\ell^2)$ time. Since the algorithm has $O(\log d)$ iterations, the overall running time is multiplied by an $O(\log d)$ factor.

We have computed only the output entries c_{id+1}. To compute c_{id+r} for other $r \in [d]$, one can shift A by $r - 1$ positions and apply the above algorithm. The final running time is multiplied by an $O(d)$ factor. □

Note that $T_{\text{select},\hat{+}}(m,d,Q) = O(T_{\text{select},+}(m,d,Q))$, since batched $A \hat{+} B$ selection reduces to batched $A + B$ selection, for example, by mapping $A_i[p]$ to $pM + A_i[p]$ and $B_j[q]$ to $-qM + B_j[q]$ for a sufficiently large M, and mapping k to $d(d-1)/2 + k$.

4.2 Putting Everything Together via 2-Level Grouping

One can further adapt Lemma 3.11 to solve not just batched $A \hat{+} B$ searching but batched $A \hat{+} B$ selection:

Lemma 4.2. *$T_{\text{select},\hat{+}}(m,d,Q) = O((T_{\text{select},\hat{+}}(\ell m, d/\ell, \ell Q) + T_{\text{sort},+}(m,d,\ell,Q) + \ell Q/w + Q + m^2 + \ell^2 m^2/\overline{w}) \cdot \log^{O(1)}(d\overline{w}) + dm \log d)$ for any given $\ell \le \min\{d, \overline{w}\}$.*

Proof. Divide each A_i into sublists $A_{i1}, \ldots, A_{i\ell}$ of size $d_0 := d/\ell$, and each B_j into sublists $B_{j1}, \ldots, B_{j\ell}$ of size d_0. Let X_{ijp} denote the list $A_{ip} + B_{jp}$ rearranged in sorted order.

The main idea is to run the algorithm select$(k, X_{ij1}, \ldots, X_{ij\ell})$, from the proof of Lemma 4.1, simultaneously for all $k \in K_{ij}$ and $i, j \in [m]$, using bit-packed lists. For each i, j, k, we maintain a list L_{ijk} of $O(\ell)$ tuples (p, k_p, s_p) of tuples for all $p \in [\ell]$ and the variables k_p and s_p kept by the algorithm. We do not explicitly store the sorted list X_{ijp}, but when desired, can retrieve the kth element in the sorted list for any given k by selection in $A_{ip} \hat{+} B_{jp}$. For line 4, we apply batched $A \hat{+} B$ selection, taking $O(T_{\text{select},\hat{+}}(\ell m, d_0, \ell Q))$ total time. For line 5, we apply batched $A + B$ sorting to first sort the tuples in L_{ijk} by μ_p and by ν_p, which would then make x easy to find; this takes $O(T_{\text{sort},+}(m,d,\ell,Q))$ total time. For line 8, we again use batched $A + B$ sorting. Other details of the bit-packed list manipulation and running-time analysis are as in the proof of Lemma 3.11. □

As before, we have $T_{\text{select},\hat{+}}(m,d,Q) = O((Q + m^2) \log^{O(1)} w)$ assuming that $d \le \delta^2 w \log m / \log^2 w$, and $w = \Omega(\log m)$, and $w \le m^{o(1)}$.

By Lemma 4.1 with $d \approx \delta^2 w \log n / \log^2 w$, one obtains:

Corollary 4.3. *(median,+)-matrix multiplication can be solved in $O((n^3/(w \log n)) \log^{O(1)} w) \le O((n^3/\log^2 n)(\log\log n)^{O(1)})$ time, and (median,+)-convolution can be solved in $O((n^2/(w \log n)) \log^{O(1)} w) \le O((n^2/\log^2 n)(\log\log n)^{O(1)})$ time, assuming that $w = \Omega(\log n)$ and $w \le n^{o(1)}$.*

5 ALGEBRAIC 3SUM

One can generalize the 3SUM algorithms to solve Barba et al.'s algebraic 3SUM problem [5] stated in the Introduction. Define $A +_\varphi B = \{\varphi(a,b) : a \in A,\ b \in B\}$. Define the batched $A +_\varphi B$ searching problem as in Problem 3.1, with $+$ replaced by $+_\varphi$, except that instead of finding the predecessor, one just wants to know whether $c \in A_i +_\varphi B_j$ for each $c \in C_{ij}$. Define the batched $A +_\varphi B$ selection/comparison/sorting problems as in Problems 3.2–3.9, with $+$ replaced by $+_\varphi$. Let $T_{\text{search},+_\varphi}(\cdot), T_{\text{select},+_\varphi}(\cdot), T_{\text{comp},+_\varphi}(\cdot), T_{\text{sort},+_\varphi}(\cdot)$ be the time complexities of these problems.

5.1 Reduction to Batched $A +_\varphi B$ Searching/Selection

Lemma 5.1. *Algebraic 3SUM can be solved in time $O(T_{\text{search},+_\varphi}(n/d, d, n^2/d) + n^2/d + n \log n)$ for any given $d \le n$.*

Proof. As in the proof of Lemma 3.3. One needs the property that for any fixed c, the number of (i,j) pairs with $c \in \varphi([A_i[1], A_{i+1}[1]) \times [B_j[1], B_{j+1}[1]))$ is $O(n/d)$ (and that these (i,j) pairs can be found in $O(n/d)$ time). In other words, a curve of the form $\{(x,y) \in \mathbb{R}^2 : \varphi(x,y) = c\}$ can intersect at most $O(n/d)$ grid cells formed by the grid lines $x = A_i[1]$ and $y = B_j[1]$. This was noted by Barba et al. [5] and follows since the curve can intersect each of the $O(n/d)$ grid lines at most $O(1)$ times. One extra case needs to be addressed: the curve may be completely contained in a grid cell without intersecting the boundary grid lines. But the number of such grid cells is $O(1)$, since the curve has $O(1)$ connected components. □

5.2 Batched $A +_\varphi B$ Selection via Cuttings in Near-logarithmic Dimensions

To adapt the proofs of Theorems 3.5 and 3.6, we replace the hyperplanes of H_j with (hyper)surfaces $\{(x_1, \ldots, x_d) \in \mathbb{R}^d : \varphi(x_u, B_j[v]) = \varphi(x_{u'}, B_j[v'])\}$ for each $u, v, u', v' \in [d]$. For these surfaces, one can apply the following version of the Cutting Lemma (Fact 3.4) with $t = 2$:

Fact 5.2 (Generalized Cutting Lemma). *Fact 3.4 remains true for a set H of n surfaces in $\mathbb{R}^d$ each of which is of the form $\{(x_1, \ldots, x_d) \in \mathbb{R}^d : (x_{u_1}, \ldots, x_{u_t}) \in S\}$ for some semi-algebraic set S of constant degree, for some $u_1, \ldots, u_t \in [d]$, and for some absolute constant t.*

Proof. Let $H_{u_1,\ldots,u_t}$ be the subset of all surfaces in H with a common tuple $(u_1, \ldots, u_t)$. For each fixed $(u_1, \ldots, u_t) \in [d]^t$, one can compute a cutting $\Gamma_{u_1,\ldots,u_t}$ into $r^{O(t)}$ cells, so that each cell is crossed by at most $|H_{u_1,\ldots,u_t}|/r$ surfaces of $H_{u_1,\ldots,u_t}$, by known results from computational geometry in a constant dimension t [2]. We output the overlay Γ of all these $O(d^t)$ cuttings, i.e., a cell in Γ is the intersection of $O(d^t)$ cells from the $\Gamma_{u_1,\ldots,u_t}$'s. Clearly, each cell in Γ is crossed by at most n/r surfaces of H.

Note that the cells of Γ are the cells in an arrangement of $O(d^t r^{O(t)})$ semi-algebraic sets in $\mathbb{R}^d$. Thus, the complexity of Γ is $O((d^t r^{O(t)})^d) \le d^{O(d)} r^{O(d)}$, assuming that t is a constant.

In the applications, one does not need an explicit representation of the cells of Γ (and they do not necessarily have constant complexity). Given the point set P, we can assign each point $p \in P$ to the cell in $\Gamma_{u_1,\ldots,u_t}$ containing p in $O(\log r)$ time by t-dimensional point location for each tuple $(u_1, \ldots, u_t)$. From this, one obtains the label of the cell in Γ containing p. □

Theorem 5.3. *$T_{\text{select},+_\varphi}(m, d, Q) = O((Q \log^2(d\overline{w}))/w + m^2/\overline{w})$ for $d \le \delta \log m / \log(\overline{w} \log m)$ for a sufficiently small constant $\delta > 0$.*

Proof. As in the proof of Theorem 3.6, using Fact 5.2. □

The analog of Theorem 3.5 without bit packing already yields an algorithm for algebraic 3SUM in $O((n^2/\log n)(\log\log n)^2)$ time, which is faster, and somewhat simpler, than Barba et al.'s algorithm. We next adapt the ideas in Section 3.3.

5.3 Batched $A +_\varphi B$ Comparisons/Sorting via Bit Packing and Fredman's Trick

Theorem 5.4. *$T_{\text{comp},+_\varphi}(m, d, Q) = O((Q \log^2 d)/w + m^2)$ for $m \ge d^{10} \log^2 d$.*

Proof. We adapt the proof of Theorem 3.8 as follows. Consider a block β of r consecutive indices in $[m]$. Build the two-dimensional arrangement $\mathcal{A}$ of $O(rd^2)$ curves $\{(x, x') \in \mathbb{R}^2 : \varphi(x, B_j[v]) = \varphi(x', B_j[v'])\}$ over all $j \in \beta$ and $v, v' \in [d]$. The arrangement has $O((rd^2)^2) = O(r^2d^4)$ cells, and one can build a data structure in $O(r^2d^4 \log^{O(1)} d)$ time supporting point location queries in $\mathcal{A}$ in $O(\log d)$ time [2]. Create a table $f : [m] \times [d]^2 \to [O(r^2d^4)]$ that maps (i, u, u') to the index of the cell of $\mathcal{A}$ that contains the point $(A_i[u], A_i[u'])$. The table can be constructed in $O(md^2 \cdot \log d)$ time by repeated point location queries. Next create a table $g : [O(r^2d^4)] \times \beta \times [d]^2 \to \{\text{true}, \text{false}\}$ that maps (γ, j, v, v') to whether $\varphi(x, B_j[v]) \le \varphi(x', B_j[v'])$ for an arbitrary point (x, x') in the cell γ of $\mathcal{A}$. The table can be constructed in $O(r^2d^4 \cdot rd^2) = O(r^3d^6)$ time. Map each tuple (u, v, u', v') in Q_{ij} to (i, j, u, v, u', v'). Concatenate the Q_{ij}'s over all $(i, j) \in [m] \times \beta$. In the concatenated list, map (i, j, u, v, u', v') to (i, u, u'), then to $f[i, u, u']$ by Fact 2.1(b); recombine with the original list to get a list of tuples $(f[i, u, u'], i, j, u, v, u', v')$, and map each of them to $(f[i, u, u'], j, v, v')$, then to $g[f[i, u, u'], j, v, v']$ by Fact 2.1(b) again; all this takes $O((Q \log^2(dm))/w + mr + md^2 + r^3d^6)$ time. One can then split the resulting list to get the answers to each Q_{ij} for all $(i, j) \in [m] \times \beta$.

Repeating the process for all $O(m/r)$ blocks β gives total time $O((Q\log^2(dm))/w + (m/r)\cdot(mr + md^2\log d + r^3d^6)) = O((Q\log^2(dm))/w + m^2)$ by setting $r = d^2\log d$, assuming that $m \geq d^{10}\log^2 d$.

As before, the $\log(dm)$ factors can be replaced by $\log d$. □

Theorem 5.5. *$T_{\text{sort},+_\varphi}(m,d,\ell,Q) = O((Q+m^2)\cdot\log^{O(1)}(dw))$ if $m \geq d^{10}\log^2 d$ and $\ell \leq \delta w/\log(dw)$ for a sufficiently small constant $\delta > 0$.*

Proof. As in the proof of Theorem 3.10. □

5.4 Putting Everything Together via 2-Level Grouping

Lemma 5.6. *$T_{\text{search},+_\varphi}(m,d,Q) = O((T_{\text{select},+_\varphi}(\ell m, d/\ell, \ell Q) + T_{\text{sort},+_\varphi}(m,d,\ell,Q) + \ell Q/w + Q + m^2 + \ell^2m^2/\overline{w})\cdot\log^{O(1)}(d\overline{w}) + dm\log d)$ for any given $\ell \leq \min\{d,\overline{w}\}$.*

Proof. As in the proof of Lemma 3.11. In algorithm search(c, A_i, B_j), replace + with φ in line 3. If $c \in A_{ip} +_\varphi B_{jq}$, then the curve $\{(x,y)\in\mathbb{R}^2 : \varphi(x,y) = c\}$ may hit the left boundary edge of $[A_{ip}[1], B_{jq}[1]) \times [A_{i,p+1}[1], B_{j,q+1}[1])$, in which case the algorithm would be correct; it may also hit one of the other three boundary edges, but each of these cases can be handled by a similar algorithm. One extra case needs to be addressed: the curve may be completely contained in a grid cell. Here, one can pick an arbitrary point in each connected component of the curve and find the unique (p,q) pair for which $[A_{ip}[1], B_{jq}[1]) \times [A_{i,p+1}[1], B_{j,q+1}[1])$ contains the point; there are only $O(1)$ extra (p,q) pairs to check for each $c \in C_{ij}$. □

Corollary 5.7. *The algebraic 3SUM problem can be solved in $O((n^2/(w\log n))\log^{O(1)} w) \leq O((n^2/\log^2 n)(\log\log n)^{O(1)})$ time, assuming that $w = \Omega(\log n)$ and $w \leq n^{o(1)}$.*

6 OFFLINE RANGE SEARCHING FOR BICHROMATIC SEGMENT INTERSECTIONS

We next adapt our 3SUM algorithms to solve the geometric problems (i)–(iv) stated in the Introduction. It suffices to consider (iv) offline triangle range searching for bichromatic segment intersections, since the other three problems reduce to it.

For simplicity, one assumes that the input is nondegenerate, for example, no three-line segments intersect at a common point, and no two intersection points have the same x-coordinate. Degeneracies can be handled by tedious modifications of our algorithms, or by applying general perturbation techniques.

We first concentrate on the special case where all line segments and triangles are *long*, i.e., have all endpoints lying on the boundary of a fixed triangle Δ_0. One may assume that the long triangles in C are halfplanes, since one can replace each long triangle with its at bounding halfplanes, so that the number of intersection points outside the triangle is equal to the sum of the number of intersection points outside the halfplanes. Without loss of generality, one may assume that the halfplanes are lower halfplanes and that all the segments in A and B touch a common edge e_0 of Δ_0 (the original problem reduces to 3 instances with this property). One may assume that e_0 is vertical.

For a point $a \in \mathbb{R}^2$, let a_x and a_y be its x- and y-coordinates, respectively. For two points $a, b \in \mathbb{R}^2$, let $\lambda(a,b)$ denote the line through a and b, i.e., $\{(x,y)\in\mathbb{R}^2 : (b_y - a_y)x + (a_x - b_x)y = b_ya_x - b_xa_y\}$. The slope of $\lambda(a,b)$ is

$$\mu(a,b) := \frac{b_y - a_y}{b_x - a_x},$$

and the x-coordinate of the intersection of $\lambda(a,b)$ and $\lambda(a',b')$ is given by the formula

$$\xi(a,b,a',b') := \frac{(a'_x - b'_x)(b_ya_x - b_xa_y) - (a_x - b_x)(b'_ya'_x - b'_xa'_y)}{(a'_x - b'_x)(b_y - a_y) - (a_x - b_x)(b'_y - a'_y)}.$$

For a segment a, let $a^* \in \mathbb{R}^2$ denote the point dual[5] to the line extension of a. For a halfplane c, let $c^* \in \mathbb{R}^2$ denote the point dual to the line bounding c. For a point a, let a^* denote the line dual to a. For a set S of points, let $S^* = \{a^* : a \in S\}$. For a set S of lines, let $\mathcal{A}(S)$ denote the arrangement of S.

For two lists A and B of long segments in $\mathbb{R}^2$, define $A \oplus B$ as the set of all intersection points between A and B.

6.1 Reduction to Batched $A \oplus B$ Searching/Selection

We replace Problems 3.1 and 3.2 with the following:

Problem 6.1 (Batched $A \oplus B$ Searching). *Given lists $A_1, \ldots, A_m, B_1, \ldots, B_m$ of d long segments each, and given "query" lists C_{ij} of lower halfplanes with $\sum_{i,j} |C_{ij}| = Q$, count the number of points in $A_i \oplus B_j$ inside c, i.e., the number of lines in $\mathcal{A}((A_i \oplus B_j)^*)$ below the point c^*, for each $c \in C_{ij}$ and each $i, j \in [m]$.*

Problem 6.2 (Batched $A \oplus B$ Selection).

(i) Given lists $A_1, \ldots, A_m, B_1, \ldots, B_m$ of d long segments each, and given "query" lists K_{ij} of numbers in $[d^4]$ with $\sum_{i,j} |K_{ij}| = Q$, find the kth leftmost vertex in $\mathcal{A}((A_i \oplus B_j)^*)$, for each $k \in K_{ij}$ and each $i, j \in [m]$.

(ii) Given sorted lists $A_1, \ldots, A_m, B_1, \ldots, B_m$ of d long segments each, and given "query" lists K_{ij} of pairs in $[d^4] \times [d^2]$ with $\sum_{i,j} |K_{ij}| = Q$, find the k'th lowest line in $\mathcal{A}((A_i \oplus B_j)^*)$ at the x-value of the kth leftmost vertex, for each $(k, k') \in K_{ij}$ and each $i, j \in [m]$.

Let $T_{\text{search},\oplus}(m, d, Q)$ and $T_{\text{select},\oplus}(m, d, Q)$ be the time complexities of Problems 6.1 and 6.2, respectively. Note that $T_{\text{search},\oplus}(m, d, Q) = O((T_{\text{select},\oplus}(m, d, Q) + Q + m^2) \cdot \log d)$ by simultaneous binary searches, first in x via Problem 6.2(i), then in y via Problem 6.2(ii), to find the level of c^* in $\mathcal{A}((A_i \oplus B_j)^*)$, for all $c \in C_{ij}$ and all i, j. (This is analogous to the standard *slab method* for planar point location [28].)

Lemma 6.3. *Offline halfplane range searching for bichromatic segment intersections in the long case can be solved in $O(T_{\text{search},\oplus}(n/d, d, n^2/d) + n^2/d + n \log n)$ time for any given $d \le n$.*

Proof. We adapt the proof of Lemma 3.3. Sort A and B by the y-values of their endpoints on e_0. As before, divide A into sublists $A_1, \ldots, A_{n/d}$ of size d, and B into sublists $B_1, \ldots, B_{n/d}$ of size d. Recall that the segments in A (respectively, B) are disjoint. Let α_i denote the region between $A_i[1]$ and $A_{i+1}[1]$ within Δ_0, and let β_j denote the region between $B_j[1]$ and $B_{j+1}[1]$ within Δ_0. For each $c \in C$, there are two types of intersections to count:

(1) intersections in $A_i \oplus B_j$ where $\alpha_i \cap \beta_j$ intersects $\partial(c \cap \Delta_0)$, and
(2) intersections in $A_i \oplus B_j$ where $\alpha_i \cap \beta_j$ is in the interior of $c \cap \Delta_0$.

For type-1 intersections, put c in C_{ij} iff α_i and β_j intersect along $\partial(c \cap \Delta_0)$. The number of such (i, j) pairs is $O(n/d)$ and they can be found in $O(n/d)$ time by a linear scan over the intersections of the segments $A_i[1]$'s and $B_j[1]$'s with $\partial(c \cap \Delta_0)$. This gives an instance of Problem 6.1 with total query list size $Q = O(n^2/d)$.

For type-2 intersections, it suffices to count the number of (i, j) pairs with $\alpha_i \cap \beta_j$ in the interior of $c \cap \Delta_0$, and multiply the number by d^2; the count can be computed in $O(n/d)$ time by another linear scan over the intersections of the segments $A_i[1]$'s and $B_j[1]$'s with $\partial(c \cap \Delta_0)$. The total time for type-2 intersections is $O(n^2/d)$. □

[5]The dual of a line with equation $y = ax - b$ refers to the point (a, b), and the dual of a point (a, b) refers to the line with equation $y = ax - b$. A point is below a line iff the dual of the point is below the dual of the line.

6.2 Batched $A \oplus B$ Selection via Cuttings in Near-logarithmic Dimensions

Theorem 6.4. *$T_{\text{select},\oplus}(m,d,Q) = O((Q\log^2(d\overline{w}))/w + m^2/\overline{w})$ for $d \le \delta \log m/\log(\overline{w}\log m)$ for a sufficiently small constant $\delta > 0$.*

Proof. To adapt the proof of Theorems 3.5 and 3.6, we redefine $p_i = (A_i[1]^*_x, A_i[1]^*_y, \ldots, A_i[d]^*_x, A_i[d]^*_y) \in \mathbb{R}^{2d}$ and redefine H_j as a set of $O(d^8)$ surfaces, containing

$$\Big\{(x_1, y_1, \ldots, x_d, y_d) \in \mathbb{R}^{2d} : \xi((x_u, y_u), B_j[v]^*, (x_{u'}, y_{u'}), B_j[v']^*) \\ = \xi((x_{u''}, y_{u''}), B_j[v'']^*, (x_{u'''}, y_{u'''}), B_j[v''']^*)\Big\}$$

for every $u, v, u', v', u'', v'', u''', v''' \in [d]$, and also

$$\Big\{(x_1, y_1, \ldots, x_d, y_d) \in \mathbb{R}^{2d} : \mu((x_u, y_u), B_j[v]^*) = \mu((x_{u'}, y_{u'}), B_j[v']^*)\Big\}$$

for every $u, v, u', v' \in [d]$,

$$\Big\{(x_1, y_1, \ldots, x_d, y_d) \in \mathbb{R}^{2d} : x_u = B_j[v]^*_x\Big\}$$

for every $u, v \in [d]$, and

$$\Big\{(x_1, y_1, \ldots, x_d, y_d) \in \mathbb{R}^{2d} : \mu((x_u, y_u), B_j[v]^*) = \mu(e^*, B_j[v]^*)\Big\}$$

for every $u, v \in [d]$ and each of the three edges e of Δ_0. Each of these surfaces can be re-expressed as $\{(x_1, y_1, \ldots, x_d, y_d) \in \mathbb{R}^{2d} : P(x_u, y_u, x_{u'}, y_{u'}, x_{u''}, y_{u''}, x_{u'''}, y_{u'''}) = 0\}$ for some degree-4 polynomial P and $u, u', u'', u''' \in [d]$, so the generalized Cutting Lemma (Fact 5.2) is applicable with $t = 8$.

Constructing the arrangement $\mathcal{A}((A_i \oplus B_j)^*)$ costs $O(d^4)$ time instead of $O(d^2 \log d)$, but this does not affect the final bound (with appropriate settings of parameters).

In Case 2, due to the definition of H_j, the sorted x-ordering of the vertices of $\mathcal{A}((A_i \oplus B_j)^*)$ is the same as the sorted x-ordering of the vertices of $\mathcal{A}((A_{i'} \oplus B_j)^*)$, and the sorted ordering of the slopes of $\mathcal{A}((A_i \oplus B_j)^*)$ is the same as the sorted ordering of the slopes of $\mathcal{A}((A_{i'} \oplus B_j)^*)$, for every $p_i, p_{i'} \in P_\Delta$. It follows that the kth leftmost vertex, or the k'th lowest line at the x-value of the kth leftmost vertex, is the same. The rest of the proof is as before. □

As before, the above theorem leads to an $O((n^2/\log n)(\log\log n)^{O(1)})$-time algorithm, which is already new. We next adapt the ideas in Section 3.3.

6.3 Batched $A \oplus B$ Comparisons/Sorting via Bit Packing and Fredman's Trick

We redefine Problem 3.7 as follows:

Problem 6.5 (Batched $A \oplus B$ Comparisons).

(i) Given lists $A_1, \ldots, A_m, B_1, \ldots, B_m$ of d long segments each, and given "query" lists Q_{ij} of tuples in $[d]^8$ with $\sum_{i,j} |Q_{ij}| = Q$, test whether $\xi(A_i[u]^*, B_j[v]^*, A_i[u']^*, B_j[v']^*) \le \xi(A_i[u'']^*, B_j[v'']^*, A_i[u''']^*, B_j[v''']^*)$ for each $(u, v, u', v', u'', v'', u''', v''') \in Q_{ij}$ and each $i, j \in [m]$.

(ii) Given lists $A_1, \ldots, A_m, B_1, \ldots, B_m$ of d long segments each, and given "query" lists Q_{ij} of tuples in $[d]^4$ with $\sum_{i,j} |Q_{ij}| = Q$, test whether $\mu(A_i[u]^*, B_j[v]^*) \le \mu(A_i[u']^*, B_j[v']^*)$ for each $(u, v, u', v') \in Q_{ij}$ and each $i, j \in [m]$.

Let $T_{\text{comp},\oplus}(m, d, Q)$ be the time complexity of the above problem.

Theorem 6.6. *$T_{\text{comp},\oplus}(m,d,Q) = O((Q\log^2 d)/w + m^2)$ for $m \ge d^{68}\log^8 d$.*

Proof. We focus on solving Problem 6.5(i), as (ii) is similar (and easier). We proceed as in the proof of Theorem 5.4. Consider a block β of r consecutive indices in $[m]$. Build the 8-dimensional arrangement $\mathcal{A}$ of $O(rd^4)$ surfaces $\{(x,y,x',y',x'',y'',x''',y''') \in \mathbb{R}^8 : \xi((x,y),B_j[v]^*,(x',y'),B_j[v']^*) = \xi((x'',y''),B_j[v'']^*,(x''',y'''),B_j[v''']^*)\}$ over all $j \in \beta$ and $v,v',v'',v''' \in [d]$. The arrangement has $O((rd^4)^8) = O(r^8 d^{32})$ cells, and one can build a data structure in $O(r^8 d^{32} \log^{O(1)} d)$ time supporting point location queries in $\mathcal{A}$ in $O(\log d)$ time [2]. Create a table $f : [m] \times [d]^4 \to [O(r^8 d^{32})]$ that maps (i,u,u',u'',u''') to the index of the cell of $\mathcal{A}$ that contains the point $(A_i[u]^*_x, A_i[u]^*_y, A_i[u']^*_x, A_i[u']^*_y, A_i[u'']^*_x, A_i[u'']^*_y, A_i[u''']^*_x, A_i[u''']^*_y)$. The table can be constructed in $O(md^4 \cdot \log d)$ time by repeated point location queries. Next create a table $g : [O(r^8 d^{32})] \times \beta \times [d]^4 \to \{\text{true}, \text{false}\}$ that maps $(\gamma, j, v, v', v'', v''')$ to whether $\xi((x,y),B_j[v]^*,(x',y'),B_j[v']^*) \le \xi((x'',y''),B_j[v'']^*,(x''',y'''),B_j[v''']^*)$ for an arbitrary point $(x,y,x',y',x'',y'',x''',y''')$ in the cell γ of $\mathcal{A}$. The table can be constructed in $O(r^8 d^{32} \cdot rd^4) = O(r^9 d^{36})$ time. Map each tuple $(u,v,u',v',u'',v'',u''',v''')$ in Q_{ij} to $(i,j,u,v,u',v',u'',v'',u''',v''')$. Concatenate the Q_{ij}'s over all $(i,j) \in [m] \times \beta$. In the concatenated list, map $(i,j,u,v,u',v',u'',v'',u''',v''')$ to (i,u,u',u'',u'''), then to $f[i,u,u',u'',u''']$ by Fact 2.1(b); recombine with the original list to get a list of tuples $(f[i,u,u',u'',u'''],i,j,u,v,u',v',u'',v'',u''',v''')$, and map each of them to $(f[i,u,u',u'',u'''],j,v,v',v'',v''')$, then to $g[f[i,u,u',u'',u'''],j,v,v',v'',v''']$ by Fact 2.1(b) again; all this takes $O(Q \log^2(dm))/w + mr + md^4 + r^9 d^{36})$ time. One can then split the resulting list to get the answers to each Q_{ij} for all $(i,j) \in [m] \times \beta$.

Repeating the process for all $O(m/r)$ blocks β gives total time $O((Q \log^2(dm))/w + (m/r) \cdot (mr + md^4 \log d + r^9 d^{36})) = O((Q \log^2(dm))/w + m^2)$ by setting $r = d^4 \log d$, assuming that $m \ge d^{68} \log^8 d$.

As before, the $\log(dm)$ factors can be replaced by $\log d$. □

Next, we redefine Problem 3.9 as follows:

Problem 6.7 (Batched $A \oplus B$ Sorting).

(i) Given lists $A_1, \ldots, A_m, B_1, \ldots, B_m$ of d long segments each, and given "query" lists Q_{ij} with and $\sum_{i,j} |Q_{ij}| = Q$, where each element of Q_{ij} is a sequence of ℓ quadruples in $[d]^4$, reorder each sequence $\langle (u_1,v_1,u'_1,v'_1), \ldots, (u_\ell,v_\ell,u'_\ell,v'_\ell) \rangle$ in Q_{ij} so that at the end, $\xi(A_i[u_1]^*,B_j[v_1]^*,A_i[u'_1]^*,B_j[v'_1]^*) \le \cdots \le \xi(A_i[u_\ell]^*,B_j[v_\ell]^*,A_i[u'_\ell]^*,B_j[v'_\ell]^*)$, for each $i,j \in [m]$.

(ii) Given lists $A_1, \ldots, A_m, B_1, \ldots, B_m$ of d long segments each, and given "query" lists Q_{ij} with $\sum_{i,j} |Q_{ij}| = Q$, where each element of Q_{ij} is a sequence containing a real number $x_0 \in \mathbb{R}$ followed by ℓ pairs in $[d]^2$, reorder each sequence $\langle x_0, (u_1,v_1), \ldots, (u_\ell,v_\ell) \rangle$ in Q_{ij} so that at the end, the lines $\lambda(A_i[u_1]^*,B_j[v_1]^*), \ldots, \lambda(A_i[u_\ell]^*,B_j[v_\ell]^*)$ are in increasing y-order at the x-value x_0, for each $i,j \in [m]$.

Let $T_{\text{sort},\oplus}(m,d,\ell,Q)$ be the time complexity of the above problem. As before, if $\ell \le \delta w/\log d$ for a sufficiently small constant δ, then the total input/output size is $O(Q + m^2)$ (since each sequence, excluding the x_0 field, can be packed in one word).

Theorem 6.8. *$T_{\text{sort},\oplus}(m,d,\ell,Q) = O((Q + m^2) \cdot \log^{O(1)}(dw))$ if $m \ge d^{68} \log^8 d$ and $\ell \le \delta w/\log(dw)$ for a sufficiently small constant $\delta > 0$.*

Proof. The batched $A \oplus B$ sorting algorithm for Problem 6.7(i) proceeds as in the proof of Theorem 3.10, using the batched $A \oplus B$ comparison subroutine for Problem 6.5(i) from Theorem 6.6.

Problem 6.7(ii) requires more effort. As before, the idea is to simulate a sorting network on all sequences in Q_{ij} for all i,j simultaneously. Consider one round of the network. Consider one sequence $\langle x_0, (u_1,v_1), \ldots, (u_\ell,v_\ell) \rangle$ in Q_{ij}. For each of the $O(\ell)$ pre-chosen index pairs (r,r') in the round, we want to test whether $\lambda(A_i[u_r]^*,B_j[v_r]^*)$ is lower than $\lambda(A_i[u_{r'}]^*,B_j[v_{r'}]^*)$

at the x-value x_0. This is true iff the following two statements are both true or both false: (a) $\xi(A_i[u_r]^*, B_j[v_r]^*, A_i[u_{r'}]^*, B_j[v_{r'}]^*) \geq x_0$; (b) $\mu(A_i[u_r]^*, B_j[v_r]^*) \leq \mu(A_i[u_{r'}]^*, B_j[v_{r'}]^*)$.

To resolve comparisons of type (a), we first sort the tuples by $\xi(A_i[u_r]^*, B_j[v_r]^*, A_i[u_{r'}]^*, B_j[v_{r'}]^*)$ via Problem 6.7(i), already solved above (for each sequence, we can extract the tuples that need to be sorted in $O(1)$ time by a word operation, since the sequence excluding x_0 is packed in a word, and the pre-chosen set of index pairs (r, r') can also be encoded in a word, as $\ell \log \ell \leq \delta w$). Then, one can resolve all the comparisons with x_0 by a standard binary search for x_0 in $O(\log \ell)$ time, for each sequence in Q_{ij} and each i, j. The cost for all these binary searches is $O(Q \log \ell)$.

To resolve comparisons of type (b), we use Problem 6.5(ii) (batched $A \oplus B$ comparisons), solved by Theorem 6.6.

Each round requires total time $O(T_{\text{comp},\oplus}(m, d, \ell Q) + (Q + m^2) \cdot \log^{O(1)}(d\ell)) = O((Q + m^2) \cdot \log^{O(1)}(d\ell))$. The running time over all rounds is multiplied by another $O(\log \ell)$ factor. □

6.4 Putting Everything Together via 2-Level Grouping

Lemma 6.9. *$T_{\text{search},\oplus}(m, d, Q) = O((T_{\text{select},\oplus}(\ell m, d/\ell, \ell Q) + T_{\text{sort},\oplus}(m, d, \ell, Q) + \ell Q/w + Q + m^2 + \ell^2 m^2/\overline{w}) \cdot \log^{O(1)}(d\overline{w}) + dm \log d)$ for any given $\ell \leq \min\{d, \overline{w}\}$.*

Proof. We adapt the proof of Lemma 3.11. If $|C_{ij}| \geq d^4$, then one can afford to use a slow algorithm: compute the arrangement $\mathcal{A}((A_i \oplus B_j)^*)$ and answer a point location query for c^* for each $c \in C_{ij}$. The number of such C_{ij}'s is at most $O(Q/d^4)$ and so the total time for this step is $O((Q/d^4) \cdot d^4 + Q \log d) = O(Q \log d)$. Thus, one may assume that $|C_{ij}| < d^4$ from now on.

Sort each A_i and B_j by the y-values of their endpoints on the edge e_0, and divide A_i into sublists $A_{i1}, \ldots, A_{i\ell}$ of size $d_0 := d/\ell$, and B_j into sublists $B_{j1}, \ldots, B_{j\ell}$ of size d_0. Let α_{ip} denote the region between $A_{ip}[1]$ and $A_{i,p+1}[1]$ within Δ_0, and let β_{jq} denote the region between $B_{jq}[1]$ and $B_{j,q+1}[1]$ within Δ_0.

Consider a fixed (i, j). For each $c \in C_{ij}$, there are two types of intersections to count:

(1) intersections in $A_{ip} \oplus B_{jq}$ where $\alpha_{ip} \cap \beta_{jq}$ intersects $\partial(c \cap \Delta_0)$, and
(2) intersections in $A_{ip} \oplus B_{jq}$ where $\alpha_{ip} \cap \beta_{jq}$ is in the interior of $c \cap \Delta_0$.

We describe an algorithm search(c, A_i, B_j) to count intersections of type 1.

For type-1 intersections, note that at least one of the four points in $A_{ip}[1] \cap \partial(c \cap \Delta_0)$ and $A_{i,p+1}[1] \cap \partial(c \cap \Delta_0)$ lies in the region β_{jq}, or at least one of the four points in $B_{jq}[1] \cap \partial(c \cap \Delta_0)$ and $B_{j,q+1}[1] \cap \partial(c \cap \Delta_0)$ lies in the region α_{ip}. Without loss of generality, assume that the left point in $A_{ip}[1] \cap \partial(c \cap \Delta_0)$ lies in the region β_{jq}; each of the other cases can be handled by a similar algorithm (we just have to be careful not to double-count). The algorithm is given by the pseudocode below, and works as follows: For each $p \in [\ell]$, first find the unique index q_p such that the left point in $A_{ip}[1] \cap \partial(c \cap \Delta_0)$ lies in the region β_{jq_p} (i.e., between $B_{jq_p}[1]$ and $B_{j,q_p+1}[1]$), by binary search (lines 2 and 3). Then find the x-rank k_p of c_x^* among the vertices in $\mathcal{A}((A_{ip} \oplus B_{jq_p})^*)$, by binary search (lines 4 and 5). Finally, find the y-rank k'_p of c_y^* among the lines in $\mathcal{A}((A_{ip} \oplus B_{jq_p})^*)$ at x-value c_x^*, by another binary search (lines 6 and 7). Then, k'_p tells us how many intersections in $A_{ip} \oplus B_{jq_p}$ lie inside c.

Search(c, A_i, B_j):
1. initialize $q_1 = \cdots = q_\ell = k_1 = \cdots = k_\ell = k'_1 = \cdots = k'_\ell = 1$
2. for $s = \ell/2, \ell/4, \ldots, 1$:
3. for each $p \in [\ell]$,
   if the left point in $A_{ip}[1] \cap \partial(c \cap \Delta_0)$ is below $B_{j,q_p+s}[1]$, then $q_p := q_p + s$
4. for $s = d_0^4/2, d_0^4/4, \ldots, 1$:

5. for each $p \in [\ell]$,
   if $c_x^* >$ (x-value of the $(k_p + s)$th leftmost vertex in $\mathcal{A}((A_{ip} \oplus B_{jq_p})^*)$), then
   $k_p := k_p + s$
6. for $s = d_0^2/2, d_0^2/4, \ldots, 1$:
7. for each $p \in [\ell]$,
   if c^* is above the $(k'_p + s)$th lowest line at the x-value of the k_pth leftmost vertex in $\mathcal{A}((A_{ip} \oplus B_{jq_p})^*)$, then $k'_p := k'_p + s$
8. for each $p \in [\ell]$, add k'_p to the count

As before, the main idea is to run search(c, A_i, B_j) simultaneously for all $c \in C_{ij}$ and all $i, j \in [m]$, using bit-packed lists.

To implement line 3, one needs to test which side of the segment c contains the point $A_{ip}[1] \cap B_{j,q_p+s}[1]$, i.e., which side of the line $\lambda(A_{ip}[1]^*, B_{j,q_p+s}[1]^*)$ contains the point c^*, for each $p \in [\ell]$. One can first sort these lines in y-order at the x-value c_x^* by batched $A \oplus B$ sorting, specifically, Problem 6.7(ii). Then, one can resolve all these line comparisons with c^* by standard binary search in y in $O(\log \ell)$ time.

To implement line 5, we apply batched $A \oplus B$ selection, namely, Problem 6.2(i), to obtain the indices to the $(k_p + s)$th leftmost vertex in $\mathcal{A}((A_{ip} \oplus B_{jq})^*)$ for each $p \in [\ell]$, and then sort these vertices by x by batched $A \oplus B$ sorting, namely, Problem 6.7(i). Then, one can resolve all the comparisons with c_x^* by standard binary search in x in $O(\log d_0)$ time.

To implement line 7, we apply batched $A \oplus B$ selection, namely, Problem 6.2(ii), to obtain the index to the $(k'_p + s)$th lowest line at the x-value of the k_pth leftmost vertex in $\mathcal{A}((A_{ip} \oplus B_{jq})^*)$ for each $p \in [\ell]$, and then sort these lines at the x-value c_x^* by batched $A \oplus B$ sorting, namely, Problem 6.7(ii). Then, one can resolve all the line comparisons with c^* by standard binary search in y in $O(\log d_0)$ time.

The details of the bit-packed list manipulation and the running-time analysis are as before.

For type-2 intersections, for a fixed (i, j) and fixed $c \in C_{ij}$, it suffices to count, for each p, the number of β_{jq}'s whose intersection with α_{ip} is in the interior of $c \cap \Delta_0$, and then multiply the total number by d^2. In other words, one wants to count the number of β_{jq}'s that are above both the left points in $A_i[1] \cap \partial(c \cap \Delta_0)$ and $A_{i+1}[1] \cap \partial(c \cap \Delta_0)$ and below both the right points in $A_i[1] \cap \partial(c \cap \Delta_0)$ and $A_{i+1}[1] \cap \partial(c \cap \Delta_0)$, or vice versa. The count can be easily computed after finding the index q of the region β_{jq} containing each of the four points in $A_i[1] \cap \partial(c \cap \Delta_0)$ and $A_{i+1}[1] \cap \partial(c \cap \Delta_0)$. Each of these four indices can be found by binary search, in a manner similar to lines 2 and 3 above. Thus, the running time is similar. □

Corollary 6.10. *Offline halfplane range searching for bichromatic segment intersections in the long case can be solved in* $O((n^2/(w \log n)) \log^{O(1)} w) \leq O((n^2/\log^2 n)(\log\log n)^{O(1)})$ *time, assuming that* $w = \Omega(\log n)$ *and* $w \leq n^{o(1)}$.

6.5 Reduction to the Long Case

Finally, one can solve the problem for arbitrary sets A and B of n disjoint (but not necessarily long) segments and an arbitrary set C of n triangles:

Corollary 6.11. *Offline triangle range searching for bichromatic segment intersections can be solved in* $O((n^2/(w \log n)) \log^{O(1)} w) \leq O((n^2/\log^2 n)(\log\log n)^{O(1)})$ *time, assuming that* $w = \Omega(\log n)$ *and* $w \leq n^{o(1)}$.

Proof. We first construct a cutting with $O(r^2)$ triangular cells, each intersecting $O(n/r)$ segments and triangle edges. The cutting and its conflict lists (list of segments and triangle edges intersecting each cell) can be generated in $O(nr)$ time [14]. We further subdivide each cell by vertical lines at each segment endpoint and triangle vertex, and we triangulate the cell (and the clipped triangles in C), so that each cell has only long segments and long triangles. The number of cells increases to $O(r^2 + n)$. We then invoke Corollary 6.10 in each cell. The total time is $O((r^2 + n) \cdot ((n/r)^2/(w \log n)) \log^{O(1)} w + nr)$.

One also needs to determine whether a triangle completely contains any cell with at least one point in $A \oplus B$. This can be done by n queries in a standard multi-level range searching structure on the $O(r^2 + n)$ cells, in $O((r^2 + n + ((r^2 + n)n)^{2/3} \log^{O(1)} n)$ time [27].

Setting $r = \sqrt{n}$ yields the result. □

7 OFFLINE REVERSE RANGE SEARCHING FOR BICHROMATIC SEGMENT INTERSECTIONS

We next adapt the algorithms in Section 6 to solve the geometric problems (v)–(vii) stated in the Introduction; the running time is slightly worse. It suffices to consider (vii) offline reverse triangle range searching for bichromatic segment intersections, since the other two problems reduce to it.

As in Section 6, we first concentrate on the special case where all line segments and triangles are long. One may again assume that the long triangles in C are halfplanes, since one can replace each long triangle with its bounding halfplanes, so that the number of long triangles not containing an intersection point $q \in \Delta_0$ is equal to the sum of the number of halfplanes not containing q.

7.1 Reduction to Batched $A \oplus B$ Searching/Selection

Let $T_{\text{search-report},\oplus}(m, d, Q)$ be the time complexity of the variant of Problem 6.1, where "count the number of" is replaced by "report all." Note that the output for each $c \in C_{ij}$ and $i, j \in [m]$ can be encoded in $O(d \log d)$ bits as a sequence of pairs of the form (u, I_u), where $u \in [d]$ and I_u is the interval of all v's such that $A_i[u] \cap B_j[v]$ lies inside c (indeed, I_u is an interval, assuming that the segments are sorted). The total output size is $O((dQ \log d)/w + Q + m^2)$. (One will not need block representations here.)

Let $T_{\text{select-report},\oplus}(m, d, Q)$ be the time complexity of the variant of Problem 6.1, where "find the k'th lowest" is replaced by "find the first k' lowest." We use the same output encoding scheme.

Like before, $T_{\text{search-report},\oplus}(m, d, Q) = O((T_{\text{select-report},\oplus}(m, d, Q) + (dQ \log d)/w + Q + m^2) \cdot \log d)$ (the $(dQ \log d)/w$ term is due to the output size).

Lemma 7.1. *Offline reverse halfplane range searching for bichromatic segment intersections in the long case can be solved in* $O(T_{\text{search-report},\oplus}(n/d, d, n^2/d) + (n^2 \log^2 d)/w + (n^2/d) \log d + n \log n)$ *time for any given* $d \leq n$.

Proof. We follow the proof of Lemma 6.3. We first describe how to compute, for each (i, j, u, v), the number of $c \in C$ that contains $A_i[u] \cap B_j[v]$ as a type-1 intersection, as defined in the proof of Lemma 6.3. These numbers will be referred to as *type-1 counters*. To this end, we define and compute the C_{ij}'s as before, and solve the reporting variant of Problem 6.1. Recall that $\sum_{i,j} |C_{ij}| = O(n^2/d)$.

Consider a fixed (i, j) with $|C_{ij}| \geq d^6$. Here, one can afford to use a slow algorithm: compute the arrangement $\mathcal{A}((A_i \oplus B_j)^*)$ in $O(d^4)$ time, answer a point location query for c^* in $O(\log d)$ time for each $c \in C_{ij}$, and count the number of points c^* inside each of the $O(d^4)$ cells of the arrangement. Since all $c \in C_{ij}$ with c^* in a common cell are equivalent, the type-1 counter for (i, j, u, v) can subsequently be computed by brute force in $O(d^4)$ time for each $u, v \in [d]$. The number of C_{ij}'s with $|C_{ij}| \geq d^6$ is at most $O((n^2/d)/d^6) = O(n^2/d^7)$, and so the total time over all i, j in this case is $O((n^2/d^7) \cdot d^2 \cdot d^4 + (n^2/d) \log d) = O((n^2/d) \log d)$.

Next consider a fixed (i, j) with $|C_{ij}| < d^6$. Here, all type-1 counters will be small (bounded by $d^{O(1)}$), so one can use bit packing to store them compactly. Sort all the $O(d|C_{ij}|)$ pairs (u, I_u)

of these elements' outputs by u in $O((d|C_{ij}|\log^2 d)/w+1)$ time (using Fact 2.1(a)) and split into sublists with a common u. For each sublist, sort the endpoints of its intervals, and perform a linear scan to determine the number of intervals containing each $v\in[d]$; these linear scans over all $u\in[d]$ take $O((d|C_{ij}|\log d)/w+d)$ time (with nonstandard word operations) and give precisely the desired counts. Since $\sum_{i,j}|C_{ij}|=O(n^2/d)$ and there are $O(n^2/d^2)$ terms, the total time over all i,j is $O((n^2\log^2 d)/w+(n^2/d^2)\cdot d)$.

Finally, we describe how to compute the number of $c\in C$ that contains $A_i[u]\cap B_j[v]$ as a type-2 intersection, for each (i,j) (the answer is independent of u and v). These numbers will be referred to as *type-2 counters*; there are only $O(n^2/d^2)$ of them. The type-2 counter for (i,j) is equal to the number of $c\in C$ such that $\alpha_i\cap\beta_j$ is strictly inside $c\cap\Delta_0$. These numbers can be generated in $O(n^2/d)$ total time over all i,j, since the number for (i,j) can be computed from that for $(i,j-1)$ by examining the halfplanes in C_{ij} and $C_{i,j-1}$.

When requested, the actual count for any given intersection point $A_i[u]\cap B_j[v]$ can be produced in constant time by summing the type-1 counter for (i,j,u,v) and the type-2 counter for (i,j). One can also determine the overall minimum/maximum count by taking the minimum/maximum type-1 counter for each (i,j) and adding to the type-2 counter for (i,j), without increasing the time bound. □

7.2 Batched $A\oplus B$ Selection via Cuttings in Near-logarithmic Dimensions

Theorem 7.2. *$T_{\text{select-report},\oplus}(m,d,Q)=O((dQ\log d)/w+Q+m^2)$ for $d\le\delta\log m/\log\log m$ for a sufficiently small constant $\delta>0$.*

Proof. As in the proof of Theorem 6.4 (except that it is sufficient to follow the simpler Theorem 3.5 instead of Theorem 3.6). The $(dQ\log d)/w$ term is due to the output size. □

It follows that $T_{\text{search-report},\oplus}(m,d,Q)=O(((dQ\log d)/w+Q+m^2)\cdot\log d)=O((Q+m^2)(\log\log m)^{O(1)})$ for $d=\delta\log m/\log\log m$, and so by Lemma 7.1, one obtains:

Corollary 7.3. *Offline reverse halfplane range searching for bichromatic segment intersections in the long case can be solved in $O((n^2/\log n)(\log\log n)^{O(1)})$ time.*

We think some small improvements are possible with a more complicated algorithm, incorporating the ideas in Sections 6.3 and 6.4, but currently we have not yet been able to achieve $O((n^2/\log^2 n)(\log\log n)^{O(1)})$ running time for this problem.

7.3 Reduction to the Long Case

Corollary 7.4. *Offline reverse triangle range searching for bichromatic segment intersections can be solved in $O((n^2/\log n)(\log\log n)^{O(1)})$ time.*

Proof. The reduction to the long case is as in the proof of Corollary 6.11. The only main change is that for each cell in the cutting, we need an additional counter for the number of triangles completely containing the cell. (When requested, the actual count for an intersection point can be found by including the additional counter for the cell containing the point to the sum.) These additional counters can be determined by answering $O(r^2+n)$ queries in a standard multi-level range searching structure on $O(n)$ triangles, in $O((n+r^2+((n+r^2)r^2)^{2/3}\log^{O(1)}n)$ time [27]. □

8 FINAL REMARKS

The analogy between 3SUM and (select,+)-convolution suggests how close in hindsight Bremner et al.'s method for (select,+)-convolution [6] was to an $O((n^2/\log n)(\log\log n)^{O(1)})$-time algorithm for 3SUM. Their method was even closer to solving the *convolution-3SUM* problem: given three lists

A, B, and C of n real numbers, decide whether there exist $i, u \in [n]$ with $C[i] = A[u] + B[i - u]$. (Bremner et al. basically solved the batched $A \hat{+} B$ selection problem with group size d near $\log n$, but convolution-3SUM reduces to batched $A \hat{+} B$ selection by simultaneous binary searches, with an extra $O(\log d) = O(\log \log n)$ factor.)

Note that (select,+)-convolution is *integer*-3SUM-hard, since the integer version of 3SUM reduces to the integer version of convolution-3SUM [26, 29] (with some extra logarithmic factor overhead), which in turn reduces to the integer version of (select,+)-convolution by simultaneous binary searches (with at most another logarithmic factor overhead).

For the geometric problems considered here, one can obtain a truly subquadratic $O(n^{2-\varepsilon})$ upper bound on the decision tree complexity, using just a subset of the ideas in Section 6. (For instance, one can just set $w \approx n^{\varepsilon}$ for a sufficiently small constant ε when working in the decision tree model, since the cost of bit manipulation is not counted; as logarithmic factors are less important here, one does not even need the part concerning cuttings in near-logarithmic dimensions.) To optimize ε in the exponent, the approach of Barba et al. [5] might be better. It is unclear if the recent techniques of Kane et al. [24] could be applicable to these geometric problems (since these problems require point location in high-dimensional arrangements of *nonlinear* surfaces, whereas Kane et al.'s technique deals with point location in arrangements of hyperplanes with integer coefficients).

We hope that my techniques will find further applications in computational geometry. Currently, the techniques are limited to geometric settings where the objects in each of the input sets A and B are disjoint. In particular, they do not work when A and B are arbitrary lines in $\mathbb{R}^2$. It remains open whether there is a subquadratic algorithm for the degeneracy testing for n lines in $\mathbb{R}^2$ (i.e., deciding whether there exist three lines meeting at a common point), or equivalently, by duality, degeneracy testing for n points in $\mathbb{R}^2$ (i.e., deciding whether there exist three collinear points). Since many 3SUM-hard problems in computational geometry are actually "collinear-triple-hard," such problems remain unaffected by our techniques at present, unfortunately.

The author's latest combinatorial algorithm for APSP or (min,+)-matrix multiplication [12] runs in $O((n^3/\log^3 n)(\log \log n)^{O(1)})$ time (building on earlier combinatorial algorithms for Boolean matrix multiplication [11, 31]), but the ideas there do not seem applicable to 3SUM or (select,+)-matrix multiplication to shave off a third logarithmic factor. Williams [30] gave a breakthrough non-combinatorial $n^3/2^{\Omega(\sqrt{\log n})}$-time algorithm for APSP or (min,+)-matrix multiplication, using the polynomial method, but these ideas have so far failed to give better results for 3SUM or (select,+)-matrix multiplication. We do not know how to obtain still faster algorithms for the integer special case of the 3SUM problem, although truly subquadratic algorithms have been found in some interesting special cases [13].

REFERENCES

[1] Pankaj K. Agarwal, Rinat Ben Avraham, Haim Kaplan, and Micha Sharir. 2014. Computing the discrete Fréchet distance in subquadratic time. *SIAM J. Comput.* 43, 2 (2014), 429–449. DOI: https://doi.org/10.1137/130920526

[2] Pankaj K. Agarwal and Micha Sharir. 2000. Arrangements and their applications. In *Handbook of Computational Geometry*. Elsevier, 49–119.

[3] Miklós Ajtai, János Komlós, and Endre Szemerédi. 1983. Sorting in $c \log n$ parallel steps. *Combinatorica* 3, 1 (1983), 1–19. DOI: https://doi.org/10.1007/BF02579338

[4] Ilya Baran, Erik D. Demaine, and Mihai Pătraşcu. 2008. Subquadratic algorithms for 3SUM. *Algorithmica* 50, 4 (2008), 584–596. DOI: https://doi.org/10.1007/s00453-007-9036-3

[5] Luis Barba, Jean Cardinal, John Iacono, Stefan Langerman, Aurélien Ooms, and Noam Solomon. 2017. Subquadratic algorithms for algebraic generalizations of 3SUM. In *Proceedings of the 33rd Symposium on Computational Geometry (SoCG'17)*. 13:1–13:15.

[6] David Bremner, Timothy M. Chan, Erik D. Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, Mihai Pătraşcu, and Perouz Taslakian. 2014. Necklaces, convolutions, and $X + Y$. *Algorithmica* 69, 2 (2014), 294–314. DOI: https://doi.org/10.1007/s00453-012-9734-3

[7] Karl Bringmann. 2014. Why walking the dog takes time: Fréchet distance has no strongly subquadratic algorithms unless SETH fails. In *Proceedings of the 55th IEEE Symposium on Foundations of Computer Science (FOCS'14)*. 661–670. DOI: https://doi.org/10.1109/FOCS.2014.76

[8] Kevin Buchin, Maike Buchin, Wouter Meulemans, and Wolfgang Mulzer. 2014. Four Soviets walk the dog—With an application to Alt's conjecture. In *Proceedings of the 25th ACM-SIAM Symposium on Discrete Algorithms (SODA'14)*. 1399–1413. DOI: https://doi.org/10.1137/1.9781611973402.103

[9] Timothy M. Chan. 2008. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica* 50, 2 (2008), 236–243. DOI: https://doi.org/10.1007/s00453-007-9062-1

[10] Timothy M. Chan. 2010. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM J. Comput.* 39, 5 (2010), 2075–2089. DOI: https://doi.org/10.1137/08071990X

[11] Timothy M. Chan. 2015. Speeding up the Four Russians algorithm by about one more logarithmic factor. In *Proceedings of the 26th ACM–SIAM Symposium on Discrete Algorithms (SODA'15)*. 212–217.

[12] Timothy M. Chan. 2017. Orthogonal range searching in moderate dimensions: k-d trees and range trees strike back. In *Proceedings of the 33rd Symposium on Computational Geometry (SoCG'17)*. 27:1–27:15.

[13] Timothy M. Chan and Moshe Lewenstein. 2015. Clustered integer 3SUM via additive combinatorics. In *Proceedings of the 47th ACM Symposium on Theory of Computing (STOC'15)*. 31–40. DOI: https://doi.org/10.1145/2746539.2746568

[14] Bernard Chazelle. 2004. Cuttings. In *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 25.1–25.10. DOI: https://doi.org/10.1201/9781420035179.ch25

[15] Mark de Berg, Joachim Gudmundsson, and Ali D. Mehrabi. 2015. Finding pairwise intersections inside a query range. In *Proceedings of the 14th Algorithms and Data Structures Symposium (WADS'15)*. 236–248. DOI: https://doi.org/10.1007/978-3-319-21840-3_20

[16] Greg N. Frederickson and Donald B. Johnson. 1982. The complexity of selection and ranking in $X+Y$ and matrices with sorted columns. *J. Comput. Syst. Sci.* 24, 2 (1982), 197–208. DOI: https://doi.org/10.1016/0022-0000(82)90048-4

[17] Greg N. Frederickson and Donald B. Johnson. 1984. Generalized selection and ranking: Sorted matrices. *SIAM J. Comput.* 13, 1 (1984), 14–30. DOI: https://doi.org/10.1137/0213002

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

[19] Ari Freund. 2017. Improved subquadratic 3SUM. *Algorithmica* 77, 2 (2017), 440–458. DOI: https://doi.org/10.1007/s00453-015-0079-6

[20] Anka Gajentaan and Mark H. Overmars. 2012. On a class of $O(n^2)$ problems in computational geometry. *Comput. Geom.* 45, 4 (2012), 140–152. DOI: https://doi.org/10.1016/j.comgeo.2011.11.006

[21] Omer Gold and Micha Sharir. 2017. Improved bounds for 3SUM, k-SUM, and linear degeneracy. In *Proceedings of the 25th European Symposium on Algorithms (ESA'17)*. 42:1–42:13. DOI: https://doi.org/10.4230/LIPIcs.ESA.2017.42

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

[23] Yijie Han. 2008. $O(n^3(\log\log n/\log n)^{5/4})$ time algorithm for all pairs shortest path. *Algorithmica* 51, 4 (2008), 428–434. DOI: https://doi.org/10.1007/s00453-007-9063-0

[24] Daniel M. Kane, Shachar Lovett, and Shay Moran. 2019. Near-optimal linear decision trees for k-SUM and related problems. *J. ACM* 66, 3 (2019), 16:1–16:18. DOI: https://doi.org/10.1145/3285953

[25] Haim Kaplan, László Kozma, Or Zamir, and Uri Zwick. 2019. Selection from heaps, row-sorted matrices, and $X+Y$ using soft heaps. In *Proceedings of the 2nd Symposium on Simplicity in Algorithms (SOSA)*. 5:1–5:21. DOI: https://doi.org/10.4230/OASIcs.SOSA.2019.5

[26] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. 2016. Higher lower bounds from the 3SUM conjecture. In *Proceedings of the 27th ACM-SIAM Symposium on Discrete Algorithms (SODA'16)*. 1272–1287. DOI: https://doi.org/10.1137/1.9781611974331.ch89

[27] Jiří Matoušek. 1992. Efficient partition trees. *Discrete Comput. Geom.* 8 (1992), 315–334. DOI: https://doi.org/10.1007/BF02293051

[28] Franco P. Preparata and Michael Ian Shamos. 1985. *Computational Geometry: An Introduction*. Springer. DOI: https://doi.org/10.1007/978-1-4612-1098-6

[29] Mihai Pătraşcu. 2010. Towards polynomial lower bounds for dynamic problems. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC'10)*. 603–610. DOI: https://doi.org/10.1145/1806689.1806772

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

[31] Huacheng Yu. 2015. An improved combinatorial algorithm for Boolean matrix multiplication. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming (ICALP'15)*. 1094–1105.

Received March 2018; revised September 2019; accepted September 2019