A Subquadratic Algorithm for 3XOR

Martin Dietzfelbinger
Technische Universität Ilmenau, Germany
martin.dietzfelbinger@tu-ilmenau.de
https://orcid.org/0000-0001-5484-3474

Philipp Schlag
Technische Universität Ilmenau, Germany
philipp.schlag@tu-ilmenau.de
https://orcid.org/0000-0001-5052-9330

Stefan Walzer
Technische Universität Ilmenau, Germany
stefan.walzer@tu-ilmenau.de
https://orcid.org/0000-0002-6477-0106

Abstract

Given a set X of n binary words of equal length w, the 3XOR problem asks for three elements $a, b, c \in X$ such that $a \oplus b = c$, where $\oplus$ denotes the bitwise XOR operation. The problem can be easily solved on a word RAM with word length w in time $O(n^2 \log n)$. Using Han's fast integer sorting algorithm (STOC/J. Algorithms, 2002/2004) this can be reduced to $O(n^2 \log \log n)$. With randomization or a sophisticated deterministic dictionary construction, creating a hash table for X with constant lookup time leads to an algorithm with (expected) running time $O(n^2)$. At present, seemingly no faster algorithms are known.

We present a surprisingly simple deterministic, quadratic time algorithm for 3XOR. Its core is a version of the PATRICIA tree for X, which makes it possible to traverse the set $a \oplus X$ in ascending order for arbitrary $a \in \{0,1\}^w$ in linear time. Furthermore, we describe a randomized algorithm for 3XOR with expected running time $O(n^2 \cdot \min\{\frac{\log^3 w}{w}, \frac{(\log \log n)^2}{\log^2 n}\})$. The algorithm transfers techniques to our setting that were used by Baran, Demaine, and Pătraşcu (WADS/Algorithmica, 2005/2008) for solving the related int3SUM problem (the same problem with integer addition in place of binary XOR) in expected time $o(n^2)$. As suggested by Jafargholi and Viola (Algorithmica, 2016), linear hash functions are employed.

The latter authors also showed that assuming 3XOR needs expected running time $n^{2-o(1)}$ one can prove conditional lower bounds for triangle enumeration just as with 3SUM. We demonstrate that 3XOR can be reduced to other problems as well, treating the examples offline SetDisjointness and offline SetIntersection, which were studied for 3SUM by Kopelowitz, Pettie, and Porat (SODA, 2016).

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases 3SUM, 3XOR, Randomized Algorithms, Reductions, Conditional Lower Time Bounds

Digital Object Identifier 10.4230/LIPIcs.MFCS.2018.59

Related Version A full version of the paper is available at [11], `http://arxiv.org/abs/1804.11086`.

43rd International Symposium on Mathematical Foundations of Computer Science (MFCS 2018).
Editors: Igor Potapov, Paul Spirakis, and James Worrell; Article No. 59; pp. 59:1–59:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The 3XOR problem [19] is the following: Given a set X of n binary strings of equal length w, are there elements $a, b, c \in X$ such that $a \oplus b = c$, where $\oplus$ is bitwise XOR? We work with the word RAM (Random Access Machine) [13] model with word length $w = \Omega(\log n)$, and we assume as usual that one input string fits into one word. Then, using sorting, the problem can easily be solved in time $O(n^2 \log n)$. Using Han's fast integer sorting algorithm [18] the time can be reduced to $O(n^2 \log\log n)$. In order to achieve quadratic running time, one could utilize a randomized dictionary for X with expected linear construction time and constant lookup time (like in [12]) or (weakly non-uniform, quite complicated) deterministic static dictionaries with construction time $O(n \log n)$ and constant lookup time as provided in [17]. Once such a dictionary is available, one just has to check whether $a \oplus b \in X$, for all pairs $a, b \in X$. No subquadratic algorithms seem to be known.

It is natural to compare the situation with that for the 3SUM problem, which is as follows:[1] Given a set X of n real numbers, are there $a, b, c \in X$ such that $a + b = c$? There is a very simple quadratic time algorithm for this problem (see Section 3 below). After a randomized subquadratic algorithm was suggested by Grønlund Jørgensen and Pettie [20], improvements ensued [14, 16], and recently Chan [8] gave the fastest deterministic algorithm known, with a running time of $n^2(\log\log n)^{O(1)}/\log^2 n$. The restricted version where the input consists of integers whose bit length does not exceed the word length w is called int3SUM. The currently best randomized algorithm for int3SUM was given by Baran, Demaine, and Pătraşcu [2, 3]; it runs in expected time $O(n^2 \cdot \min\{\frac{\log^2 w}{w}, \frac{(\log\log n)^2}{\log^2 n}\})$ for $w = O(n \log n)$. The 3SUM problem has received a lot of attention in recent years, because it can be used as a basis for conditional lower time bounds for problems, for instance, from computational geometry and data structures [15, 22, 26]. Because of this property, 3SUM is in the center of attention of papers dealing with low-level complexity. Chan and Lewenstein [9] give upper bounds for inputs with a certain structure. Kane, Lovett, and Moran [21] prove near-optimal upper bounds for linear decision trees. Wang [28] considers randomized algorithms for subset sum, trying to minimize the space, and Lincoln et al. [23] investigate time-space tradeoffs in deterministic algorithms for k-SUM. Barba et al. [4] examine a generalization of 3SUM in which the sum function is replaced by a constant-degree polynomial in three variables. Chan [7] shows how to adapt the ideas of the subquadratic int3SUM algorithm to the general position problem.

In contrast, 3XOR received relatively little attention, before Jafargholi and Viola [19] studied 3XOR and described techniques for reducing this problem to triangle enumeration. In this way they obtained conditional lower bounds in a way similar to the conditional lower bounds based on int3SUM.

The main results of this paper are the following: (1) We present a surprisingly simple deterministic algorithm for 3XOR that runs in time $O(n^2)$ (Theorem 5). When X is given in sorted order, it constructs in linear time a version of the PATRICIA tree [25] for X, using only word operations and not looking at single bits at all. This tree then makes it possible to traverse the set $a \oplus X$ in ascending order in linear time, for arbitrary $a \in \{0,1\}^w$. This is sufficient for achieving running time $O(n^2)$. (2) The second result is a randomized algorithm for 3XOR that runs in time $O(n^2 \cdot \min\{\frac{\log^3 w}{w}, \frac{(\log\log n)^2}{\log^2 n}\})$ for $w = O(n \log n)$ (Theorem 7),

[1] There are many different, but equivalent versions of 3SUM and 3XOR, differing in the way the input elements are grouped. Often one sees the demand that the three elements a, b, and c with $a \oplus b = c$ or $a + b = c$, resp., come from different sets.

which is almost the same bound as that of [2] for int3SUM. Finding a deterministic algorithm for 3XOR with subquadratic running time remains an open problem. (3) Finally, we reduce 3XOR to offline SetDisjointness (Theorem 10) and offline SetIntersection (Theorem 11), establishing the conditional lower bound $n^{2-o(1)}$ (as in [22] conditioned on the int3SUM conjecture).

Unfortunately, no (non-trivial) relation between the required (expected) time for 3SUM and 3XOR is known. In particular, we cannot exclude the case that one of these problems can be solved in (expected) time $O(n^{2-\varepsilon})$ for some constant $\varepsilon > 0$ whereas the other one requires (expected) time $n^{2-o(1)}$. Actually, this possibility is the background of some conditional statements on the cost of listing triangles in graphs in [19, Cor. 2]. However, due to the similarity of 3XOR to 3SUM, the question arises whether the recent results on 3SUM can be transferred to 3XOR.

In Section 2, we review the word RAM model and examine 1-universal classes of linear hash functions. In particular, we determine the evaluation cost of such hash functions and we restate a hashing lemma [2] on the expected number of elements in "overfull" buckets. Furthermore, we state how fast one can solve the set intersection problem on word-packed arrays. In Section 3, we construct a special enhanced binary search tree T_X to represent a set X of binary strings of fixed length. This representation makes it possible to traverse the set $a \oplus X$ in ascending order for any $a \in \{0,1\}^w$ in linear time, which leads to a simple deterministic algorithm for 3XOR that runs in time $O(n^2)$. Then, we turn to randomized algorithms and show how to solve 3XOR in subquadratic expected time in Section 4: $O(n^2 \cdot \min\{\frac{\log^3 w}{w}, \frac{(\log\log n)^2}{\log^2 n}\})$ for $w = O(n \log n)$, and $O(n \log^2 n)$ for $n \log n \leq w = O(2^{n \log n})$. Our approach uses the ideas of the subquadratic expected time algorithm for int3SUM presented in [2], i. e., computing buckets and fingerprints, word packing, exploiting word-level parallelism, and using lookup tables to solve the set intersection problem on word-packed arrays. Altogether, we get the same expected running time for $w = O(\log^2 n)$ and a word-length-dependent upper bound on the expected running time for $w = \omega(\log^2 n)$ that is worse by a $\log w$ factor in comparison to the int3SUM setting. Based on these results and the similarity of 3XOR to int3SUM, it seems natural to conjecture that 3XOR requires expected time $n^{2-o(1)}$, too, and so 3XOR is a candidate for reductions to other computational problems just as 3SUM. In Section 5, we describe how to reduce 3XOR to offline SetDisjointness and offline SetIntersection, transferring the results of [22] from int3SUM to 3XOR.

Recently, Bouillaguet et al. [6] studied algorithms "for the 3XOR problem". This is related to our setting, but not identical. These authors study a variant of the "generalized birthday problem", well known in cryptography as a problem to which some attacks on cryptosystems can be reduced, see [6]. Translated into our notation, their question is: Given a *random* set $X \subseteq \{0,1\}^w$ of size $3 \cdot 2^{w/3}$, find, if possible, three different strings $a, b, c \in X$ such that $a \oplus b = c$. Adapting the algorithm from [2], these authors achieve a running time of $O(2^{2w/3}(\log^2 w)/w^2)$, which corresponds to the running time of our algorithm for $n = 3 \cdot 2^{w/3}$. The difference to our situation is that their input is random. This means that the issue of 1-universal families of linear hash functions disappears (a projection of the elements in X on some bit positions does the job) and that complications from weak randomness are absent (e. g., one can use projection into relatively small buckets and use Chernoff bounds to prove that the load is very even with high probability). This means that the algorithm described in [6] does not solve our version of the 3XOR problem.

2 Preliminaries

2.1 The Word RAM Model

As is common in the context of fast algorithms for the int3SUM problem [2], we base our discussion on the *word RAM model* [13]. This is characterized by a word length w. Each memory cell can store w bits, interpreted as a bit string or an integer or a packed sequence of subwords, as is convenient. The word length w is assumed to be at least $\log n$ and at least the bit length of a component of the input. It is assumed that the operations of the *multiplicative instruction set*, i. e., arithmetic operations (addition, subtraction, multiplication), word operations (left shift, right shift), bitwise Boolean operations (AND, OR, NOT, XOR), and random memory accesses can be executed in constant time. We will write $\oplus$ to denote the bitwise XOR operation. A *randomized* word RAM also provides an operation that in constant time generates a uniformly random value in $\{0, 1, \ldots, v-1\}$ for any given $v \leq 2^w$.

2.2 Linear Hash Functions

We consider hash functions $h\colon U \to M$, where the domain ("universe") U is $\{0,1\}^\ell$ and the range M is $\{0,1\}^\mu$ with $\mu \leq \ell$. Both universe and range are vector spaces over $\mathbb{Z}_2$. In [2] and in successor papers on int3SUM "almost linear" hash functions based on integer multiplication and truncation were used, as can be found in [10]. As noted in [19], in the 3XOR setting the situation is much simpler. We may use $\mathcal{H}^{\mathrm{lin}}_{\ell,\mu}$, the set of all $\mathbb{Z}_2$-linear functions from U to M. A function h_A from this family is described by a $\mu \times \ell$ matrix A, and given by $h_A(x) = A \cdot x$, where $x = (x_0, \ldots, x_{\ell-1})^\intercal \in U$ and $h_A(x) \in M$ are written as column vectors. For all hash functions $h \in \mathcal{H}^{\mathrm{lin}}_{\ell,\mu}$ and all $x, y \in U$ we have $h(x \oplus y) = h(x) \oplus h(y)$, by the very definition of *linearity*. Further, this family is 1-*universal*, indeed, we have $\mathbf{Pr}_{A \in \{0,1\}^{\mu\times\ell}}[\, h_A(x) = h_A(y)\,] = \mathbf{Pr}_{A \in \{0,1\}^{\mu\times\ell}}[\, h_A(x \oplus y) = 0\,] = 2^{-\mu} = 1/|M|$, for all pairs x, y of different keys in U. We remark that the convolution class described in [24], a subfamily of $\mathcal{H}^{\mathrm{lin}}_{\ell,\mu}$, can be used as well, as it is also 1-universal, and needs only $\ell + \mu - 1$ random bits.

The universe we consider here is $\{0,1\}^w$. The time for evaluating a hash function $h \in \mathcal{H}^{\mathrm{lin}}_{w,\mu}$ on one or on several inputs depends on the instruction set and on the way $h = h_A$ is stored. In contrast to the int3SUM setting [2], we are not able to calculate hash values in constant time.

▶ **Lemma 1.** *For $h \in \mathcal{H}^{\mathrm{lin}}_{w,\mu}$ and inputs from $\{0,1\}^w$ we have:*

(a) *$h(x)$ can be calculated in time $O(\mu)$, if* PARITY *of w-bit words is a constant time operation.*

(b) *$h(x)$ can always be calculated in time $O(\mu + \log w)$.*

(c) *$h(x_1), \ldots, h(x_n)$ can be evaluated in time $O(n\mu + \log w)$.*

Proof. (Sketch.) Assume $h = h_A$. For (a) we store the rows of A as w-bit strings, and obtain each bit of the hash value by a bitwise $\wedge$ operation followed by PARITY. For (b) we assume the w columns of A are stored as μ-bit blocks, in $O(\mu)$ words. An evaluation is effected by selecting the columns indicated by the 1-bits of x and calculating the $\oplus$ of these vectors in a word-parallel fashion. In $\log w$ rounds, these vectors are added, halving the number of vectors in each round. For (c), we first pack the columns selected for the n input strings into $O(n\mu)$ words and then carry out the calculation indicated in (b), but simultaneously for all x_i and within as few words as possible. This makes it possible to further exploit word-level parallelism, if μ should be much smaller than w. ◀

We shall use linear, 1-universal hashing for splitting the input set into buckets and for replacing keys by fingerprints in Section 4.

▶ Remark. In the following, we will apply Lemma 1(c) to map n binary strings of length w to hash values of length $\mu = O(\log n)$ in time $O(n \log n + \log w)$. Since $\log w$ will dominate the running time only for huge word lengths, we assume in the rest of the paper that $w = 2^{O(n \log n)}$ and that all hash values can be calculated in time $O(n \log n)$.

▶ Remark. When randomization is allowed, we will assume that we have constructed in expected $O(n)$ time a standard hash table for input set X with constant lookup time [12]. (Arbitrary 1-universal classes can be used for this.)

2.3 A Hashing Lemma for 1-Universal Families

A hash family $\mathcal{H}$ of functions from U to M is called *1-universal* if $\mathbf{Pr}_{h \in \mathcal{H}}[\, h(x) = h(y) \,] \leq 1/|M|$ for all $x, y \in U$, $x \neq y$. We map a set $S \subseteq U$ with $|S| = n$ into M with $|M| = m$ by a random element $h \in \mathcal{H}$. In [2, Lemma 4] it was noted that for 1-universal families the expected number of keys that collide with more than $3n/m$ other keys is bounded by $O(m)$. We state a slightly stronger version of that lemma. (The strengthening is not essential for the application in the present paper.)

▶ **Lemma 2** (slight strengthening of Lemma 4 in [2]). *Let $\mathcal{H}$ be a 1-universal class of hash functions from U to M, with $m = |M|$, and let $S \subseteq U$ with $|S| = n$. Choose $h \in \mathcal{H}$ uniformly at random. For $i \in M$ define $B_i = \{y \in S \mid h(y) = i\}$. Then for $2\frac{n}{m} < t \leq n$ we have:*

$$\mathbf{E}_{h \in \mathcal{H}}[\, |\{x \in S \mid |B_{h(x)}| \geq t\}| \,] < \frac{n}{t - 2\frac{n}{m}}$$

(The bound in [2] was about twice as large. The proof is given in the full version [11].)

In our algorithm, we will be interested in the number of elements in buckets with size at least three times the expectation. Choosing $t = 3\frac{n}{m}$ in Lemma 2, we conclude that the expected number of such elements is smaller than the number of buckets.

▶ **Corollary 3.** *In the setting of Lemma 2 we have $\mathbf{E}_{h \in \mathcal{H}}[\ |\{\, x \in S \ \mid \ |B_{h(x)}| \geq 3n/m \,\}| \ ] < m$.*

2.4 Set Intersection on Unsorted Word-Packed Arrays

We consider the problem "*set intersection on unsorted word-packed arrays*": Assume k and ℓ are such that $k(\ell + \log k) \leq w$, and that two words a and b are given that both contain k many ℓ-bit strings: a contains $a_0, \ldots, a_{k-1}$ and b contains $b_0, \ldots, b_{k-1}$. We wish to determine whether $\{a_0, \ldots, a_{k-1}\} \cap \{b_0, \ldots, b_{k-1}\}$ is empty or not and find an element in the intersection if it is nonempty.

In [3, proof of Lemma 3] a similar problem is considered: It is assumed that a is sorted and b is *bitonic*, meaning that it is a cyclic rotation of a sequence that first grows and then falls. In this case one sorts the second sequence by a word-parallel version of bitonic merge (time $O(\log k)$), and then merges the two sequences into one sorted sequence (again in time $O(\log k)$). Identical elements now stand next to each other, and it is not hard to identify them. We can use a slightly slower modification of the approach of [3]: We sort both sequences by word-packed bitonic sort [1] (simulating Batcher's *bitonic sort* sorting network [5] on a word-packed array), which takes time $O(\log^2 k)$, and then proceed as before.[2] We obtain the following result.

[2] It is this slower version of packed intersection that causes our randomized 3XOR algorithm to be a little slower than the int3SUM algorithm for $w = \omega(\log^2 n)$.

Algorithm 1: A simple quadratic 3SUM algorithm.

```
Algorithm 3SUM(X):
    sort X as x_1 < ··· < x_n
    for a ∈ X do
        (i, j) ← (1, 1)
        while i ≤ n AND j ≤ n do
            if a + x_i < x_j then
                i ← i + 1
            else if a + x_i > x_j then
                j ← j + 1
            else return (a, x_i, x_j)
    return no solution
```

Algorithm 2: A quadratic 3XOR algorithm.

```
Algorithm 3XOR(X):
    sort X as x_1 < ··· < x_n
    T_X ← makeTree(X)
    for a ∈ X do
        (i, j) ← (1, 1)
        (y_i)_{1≤i≤n} ← traverse(T_X, a)
        while i ≤ n AND j ≤ n do
            if y_i < x_j then
                i ← i + 1
            else if y_i > x_j then
                j ← j + 1
            else return (a, y_i ⊕ a, x_j)
    return no solution
```

▶ **Lemma 4.** *Assume $k(\ell + \log k) = O(w)$, and assume that two sequences of ℓ-bit strings, each of length k, are given. Then the t entries that occur in both sequences can be listed in time $O(\log^2 k + t)$.*

A more detailed description is given in the full version [11].

3 A Deterministic 3XOR Algorithm in Quadratic Time

A well known deterministic algorithm for solving the 3SUM problem in time $O(n^2)$ is reproduced in Algorithm 1. After sorting the input X as $x_1 < \cdots < x_n$ in time $O(n \log n)$, we consider each $a \in X$ separately and look for triples of the form $a + b = c$. Such triples correspond to elements of the intersection of $a + X = \{a + x_1, \ldots, a + x_n\}$ and X. Since X is sorted, we can iterate over both X and $a + X$ in ascending order and compute the intersection with an interleaved linear scan.

Unfortunately, the $\oplus$-operation is not order preserving, indeed, $x < y$ does not imply $a \oplus x < a \oplus y$ for the lexicographic ordering on bitstrings – or, indeed, any total ordering on bitstrings. We may sort X and each set $a \oplus X = \{a \oplus x \mid x \in X\}$, for $a \in X$, separately to obtain an algorithm with running time $O(n^2 \log n)$. Using fast deterministic integer sorting [18] reduces this to time $O(n^2 \log \log n)$. In order to achieve quadratic running time, one may utilize a randomized dictionary for X with expected linear construction time and constant lookup time (like in [12]) or (weakly non-uniform, rather complex) deterministic static dictionaries with construction time $O(n \log n)$ and constant lookup time as provided in [17]. Once such a dictionary is available, one just has to check whether $a \oplus b \in X$, for all $a, b \in X$.

Here we describe a rather simple deterministic algorithm with quadratic running time. For this, we utilize a special binary search tree[3] T_X that allows, for arbitrary $a \in \{0,1\}^w$, to traverse the set $a \oplus X = \{a \oplus x \mid x \in X\}$ in lexicographically ascending order, in linear time. For $X \neq \emptyset$, the tree T_X is recursively defined as follows.

[3] The structure of the tree is that of the PATRICIA tree [25] for X.

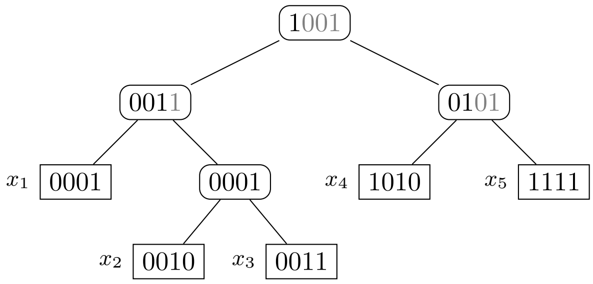

Figure 1 The tree T_X for $X = \{x_1 = 0001, x_2 = 0010, x_3 = 0011, x_4 = 1010, x_5 = 1111\}$. The first 1-bit of the label of an inner node indicates the most significant bit that is not constant among the x-values managed by that subtree (the bits after the first 1-bit are irrelevant). According to the value of this bit, elements are found in the left or right subtree. Apart from the labels of the inner nodes, T_X is essentially the PATRICIA tree [25] for X.

- If $X = \{x\}$, then T_X is $\mathsf{LeafNode}(x)$, a tree consisting of a single leaf with label x.
- If $|X| \geq 2$, let $\text{lcp}(X)$ denote the longest common prefix of the elements of X when viewed as bitstrings. That is, all elements of X coincide on the first $k = |\text{lcp}(X)|$ bits, the elements of some nonempty set $X_0 \subsetneq X$ start with $\text{lcp}(X)0$ and the elements of $X_1 = X - X_0$ start with $\text{lcp}(X)1$. We define $T_X = \mathsf{InnerNode}(T_{X_0}, 0^k 1b, T_{X_1})$ for some $b \in \{0,1\}^{w-k-1}$, meaning that T_X consists of a root vertex with label $\ell = 0^k 1b$, a left subtree T_{X_0} and a right subtree T_{X_1}. The choice of b is irrelevant, but it is convenient to define the label more concretely as $\ell = (\max X_0) \oplus (\min X_1)$.

Note that along paths of inner nodes down from the root the labels when regarded as integers are strictly decreasing. We give an example in Figure 1 and provide a $O(n \log n)$ time construction of T_X from X in Algorithm 4.

In the context of $T_X = \mathsf{InnerNode}(T_{X_0}, \ell = 0^k 1b, T_{X_1})$ as described above, the $(k+1)$st bit is the most significant bit where elements of X differ. Crucially, this is also true for the set $a \oplus X$ for any $a \in \{0,1\}^w$. Since the elements of X are partitioned into X_0 and X_1 according to their $(k+1)$st bit, either all elements of $a \oplus X_0$ are less than all elements of $a \oplus X_1$, or vice versa, depending on whether the $(k+1)$st bit of a is 0 or 1. Using that the $(k+1)$st bit of a is 1 iff $a \oplus \ell < a$, this suggests a simple recursive algorithm to produce $a \oplus X$ in sorted order, given as Algorithm 3.

With the data structure T_X in place, the strategy from 3SUM carries over to 3XOR as seen in Algorithm 2. Summing up, we have obtained the following result:

▶ **Theorem 5.** *A deterministic word RAM can solve the* 3XOR *problem in time* $O(n^2)$. ◀

In Algorithm 4 we provide a linear time construction of T_X from a stream containing the sorted array X interleaved with the labels $\ell_i = x_i \oplus x_{i+1}$ (due to sorting the total runtime is $O(n \log n)$). Despite its brevity, the recursive build function is somewhat subtle.

▶ **Claim 6** (Correctness of Algorithm 4). *If build() is called while the stream contains the elements* $(\ell_i, x_{i+1}, \ldots, x_n, \ell_n = \infty)$*, the call consumes a prefix of the stream until* $\mathsf{top}(\text{stream}) = \ell_j$ *where* $j = \min\{j > i \mid \ell_j \geq \ell_i\}$*. It returns* T_X *where* $X = \{x_{i+1}, \ldots, x_j\}$*.*

Once this is established, the correctness of makeTree immediately follows as for the outer call we have $i = 0$ and $j = n$ (with the understanding that $\infty \geq \infty$).

Proof of Claim 6. By the ℓ-call we mean the (recursive) call to build() with $\mathsf{top}(\text{stream}) = \ell$. In particular the ℓ-call consumes ℓ from the stream and our claim concerns the ℓ_i-call. It is clear from the algorithm that an ℓ-call can only invoke an ℓ'-call if $\ell' < \ell$. Therefore the

Algorithm 3: Given a tree $T = T_X$ and $a \in \{0,1\}^w$, the algorithm yields the elements of $a \oplus X = \{a \oplus x \mid x \in X\}$ in sorted order.

Algorithm traverse(T, a):
  if $T = \mathsf{LeafNode}(x)$ **then**
    yield $a \oplus x$
  else
    let $T = \mathsf{InnerNode}(T_0, \ell, T_1)$
    if $a \oplus \ell > a$ **then**
      traverse(T_0, a)
      traverse(T_1, a)
    else
      traverse(T_1, a)
      traverse(T_0, a)

Algorithm 4: $O(n \log n)$-time algorithm to construct T_X from X.

Algorithm makeTree(X)**:**
  sort X as $x_1 < \cdots < x_n$
  let $\ell_i = x_i \oplus x_{i+1}, \quad 1 \leq i < n$
  stream $\leftarrow$ $(\infty, x_1, \ell_1, \ldots, \ell_{n-1}, x_n, \infty)$
  return build() where
  subroutine build()**:**
    $\ell \leftarrow$ pop(stream)
    $x \leftarrow$ pop(stream)
    $T \leftarrow \mathsf{LeafNode}(x)$
    while top(stream) $< \ell$ **do**
      $\ell' \leftarrow$ top(stream)
      $T \leftarrow \mathsf{InnerNode}(T, \ell', \text{build}())$
    return T

ℓ_i-call cannot directly or indirectly cause the ℓ_j-call since $\ell_j \geq \ell_i$. At the same time, the ℓ_i-call can only terminate when $\mathsf{top}(\mathit{stream}) \geq \ell_i$. This establishes that $\ell_j = \mathsf{top}(\mathit{stream})$ when the ℓ_i-call ends – the first part of our claim.

Next, note that since X is sorted, there is some m such that we have $X_0 = \{x_{i+1}, \ldots, x_m\}$ and $X_1 = \{x_{m+1}, \ldots, x_j\}$ where $X = X_0 \cup X_1$ is the partition from the definition of T_X. Moreover, ℓ_m is the largest label among $\ell_{i+1}, \ldots, \ell_{j-1}$. This implies that the ℓ_m-call is *directly* invoked from the ℓ_i-call. Just before the ℓ_m-call is made, the ℓ_i-call played out just as though the stream had been $(\ell_i, x_{i+1}, \ldots, x_m, \ell'_m = \infty)$, which would have produced T_{X_0} by induction[4]. However, due to $\ell_m = \mathsf{top}(\text{stream}) < \ell = \ell_i$, instead of returning $T = T_{X_0}$, the while loop is entered (again) and produces $\mathsf{InnerNode}(T = T_{X_0}, \ell = \ell_m, \text{build}())$. The stream for the ℓ_m-call is $(\ell_m, \ldots, x_n, \ell_n)$ and ℓ_j is the first label not smaller than ℓ_m. So, again by induction, the ℓ_m-call produces T_{X_1} and ends with $\mathsf{top}(\text{stream}) = \ell_j$. Given this, it is clear that afterwards the loop condition in the ℓ_i-call is not satisfied (since $\ell_j \geq \ell_i$) and the new $T = T_X$ is returned immediately, establishing the second part of the claim. ◀

4 A Subquadratic Randomized Algorithm

In this section we present a subquadratic expected time algorithm for the 3XOR problem. Its basic structure is the same as in the corresponding algorithm for int3SUM presented in [2]. We use 1-universal families of linear hash functions to split the elements into buckets and to compute short fingerprints. Due to linearity, the bucket of an element c with $c = a \oplus b$ is uniquely determined when knowing the buckets of a and b. Furthermore, if $a \oplus b = c$, then this equation is also true when looking at the fingerprints of these elements. Therefore, packing the fingerprints of all elements of a (not too full) bucket into one word (a word-packed array) allows us to evaluate the latter equation for a lot of triples "in parallel". For this

[4] Formally the induction is on the value of $j - i$. The case of $j - i = 1$ is trivial.

purpose, we exploit word-level parallelism and use lookup tables to solve the set intersection problem on unsorted word-packed arrays. Algorithm 5 illustrates this strategy for large word lengths $w = \Omega((\log^2 n) \log\log n)$.

Changes (in comparison to the int3SUM algorithm) are made where necessary to deal with the different setting. This makes it a little more difficult in some parts of the algorithm (mainly because XOR-ing a sorted sequence with some a will destroy the order) and easier in other parts (in particular where linearity of hash functions is concerned). Altogether, we get an expected running time that is the same as in [2] for $w = O(\log^2 n)$ and slightly worse for larger w. Recall we assume $w = 2^{O(n \log n)}$ throughout.

▶ **Theorem 7.** *A randomized word RAM with word length w can solve the* 3XOR *problem in expected time*

$$O\left(n^2 \cdot \min\left\{\frac{\log^3 w}{w}, \frac{(\log\log n)^2}{\log^2 n}\right\}\right) \quad \text{for } w = O(n \log n),$$

and $O(n \log^2 n)$, otherwise.

The crossover point between the w and the $\log n$ factor is $w = (\log^2 n) \log\log n$. The only difference to the running time of [2] is in an extra factor $\log w$ in the word-length-dependent part.

Proof. We describe the main ideas of the algorithm. For full details, see [11]. If $w = \omega(n \log n)$, we proceed as for $w = \Theta(n \log n)$. We use two levels of hashing.

Good and Bad Buckets. We split X into $R = 2^r = o(n)$ *buckets* $X_u = \{x \in X \mid h_1(x) = u\}$, $u \in \{0,1\}^r$, using a randomly chosen hash function $h_1 \in \mathcal{H}^{\text{lin}}_{w,r}$. The hash values are calculated once and for all and stored for further use. By linearity, for every solution $a \oplus b = c$ we also have $h_1(a) \oplus h_1(b) = h_1(c)$. Given $a \in X_u$ and $b \in X_v$, we only have to inspect bucket $X_{u \oplus v}$ when looking for a $c \in X$ such that $a \oplus b = c$.

For $a \in X$, the expected size of bucket $X_{h_1(a)}$ is n/R. A bucket of size larger than $3n/R$ is called *bad*, as are elements of bad buckets. All other buckets and elements are called *good*. By Corollary 3, the expected number of bad elements is smaller than R. We can even assume that the total number of bad elements is smaller than $2R$. (By Markov's inequality, we simply have to repeat the choice of h_1 expected $O(1)$ times until this condition is satisfied.)

Fingerprints and Word-Packed Arrays. Furthermore, we use another hash function $h_2 \in \mathcal{H}^{\text{lin}}_{w,p}$ for some appropriately chosen p to calculate p-bit *fingerprints* for all elements in X. If $(3n/R) \cdot p \le w$, we can pack all fingerprints of elements of a good bucket X_u into one word X^*_u. This packed representation is called *word-packed array*. Again by linearity, for every solution $a \oplus b = c$ we have $h_2(a) \oplus h_2(b) = h_2(c)$. On the other hand, the expected number of *colliding triples*, i.e., triples with $a \oplus b \ne c$ but $h_1(a) \oplus h_1(b) = h_1(c)$ and $h_2(a) \oplus h_2(b) = h_2(c)$, is at most $2n^3/(R \cdot 2^p)$. (Since the hash families that we use are 1-universal, the probability that a triple with $a \oplus b \ne c$ is colliding is at most $2/(R \cdot 2^p)$. The additional factor 2 is due to the repeated choice of h_1 until there are fewer than $2R$ bad elements.)

The total time for all the hashing steps described so far is $O(n \cdot (r + p))$, see Section 2.2. We consider two choices of $R = 2^r$ and p, cf. [2, proof of Lemma 3] and [2, proof of Thm. 2]. The first one is better for larger words of length $w = \Omega((\log^2 n) \log\log n)$ whereas the second one yields better results for smaller words. In both cases, we search for triples with a fixed number of bad elements separately. The strategies for finding triples of good elements correspond to the approach for int3SUM in [2]. However, for triples with at least one bad element we have to rely on a more fine-grained examination than in [2]. For this, we will use hash tables and another lookup table.

Long Words: Exploiting Word-Level Parallelism. If the word length is large enough, i. e., $w = \Omega((\log^2 n) \log\log n)$, we choose $R \approx \lceil 6 \cdot n \cdot (\log w)/w \rceil$ as a power of 2 and $p = \lfloor 2 \cdot \log w \rfloor$ to be able to pack all fingerprints of elements of a good bucket into one word. We examine triples with at most one and at least two bad elements separately, as seen in Algorithm 5.

When looking for triples with at most one bad element, we do the following for every (good or bad) $a \in X$ and $u \in \{0,1\}^r$ where X_u and the corresponding bucket $X_{h_1(a) \oplus u}$ are good (as in [2, proof of Lemma 3] when examining triples of three good elements): We XOR every fingerprint of the word-packed array X_u^* with $h_2(a)$. Then, in time $O(\log^2 (n/R) + t) = O(\log^2 w + t)$, we construct a list of t common pairs in this modified word-packed array and $X^*_{h_1(a) \oplus u}$, which is possible by Lemma 4. For each such pair, we only have to check whether it derives from a non-colliding triple. Since we can stop when we find a non-colliding triple and since the expected total number of colliding triples is $O(n^2/(w \log w))$, we are done in expected time $O(n \cdot R \cdot \log^2 w + n^2/(w \log w)) = O(n^2(\log^3 w)/w)$.

In order to examine all triples with at least two bad elements, we provide a hash table for X with expected construction time $O(n)$ and constant lookup time [12]. Now, for each of the at most $4R^2 = O(n^2(\log^2 w)/w^2)$ pairs (a, b) of bad elements we can check if $a \oplus b \in X$ in constant time.[5]

The total expected running time for this parameter choice is $O(n^2(\log^3 w)/w)$.

Short Words: Using Lookup Tables. For word lengths $w = O((\log^2 n) \log\log n)$, we choose $R \approx \lceil 55 \cdot n \cdot (\log\log n)/\log n \rceil$ as a power of 2 and $p = \lfloor 6 \cdot \log\log n \rfloor$ to pack all fingerprints of elements of a good bucket into $(\frac{1}{3} - \varepsilon)\log n$ bits, for some $\varepsilon > 0$.

We start by looking for triples with no bad element. For this, we consider all $\leq R^2$ triples of corresponding good buckets (as in [2, proof of Thm. 2]). We use a lookup table of size $n^{1-\Omega(1)}$ to check whether such a triple of buckets yields a triple of fingerprints (in the word-packed arrays) with $h_2(a) \oplus h_2(b) = h_2(c)$ in constant time. If this is the case, we search for a corresponding triple $a \oplus b = c$ in the buckets of size $O((\log n)/\log\log n)$ in time $O((\log^3 n)/(\log\log n)^3)$. Since one table entry can be computed in time $O((\log^3 n)/(\log\log n)^3)$, setting up the lookup table takes time $n^{1-\Omega(1)}$. Furthermore, the expected $O(n^2/((\log\log n) \log^5 n))$ colliding triples cause additional expected running time $O(n^2/((\log\log n)^4 \log^2 n))$. Since we can stop when we find a non-colliding triple, the total expected time is $O(R^2) = O(n^2(\log\log n)^2/\log^2 n)$.

Searching for triples with exactly one bad element can be done in a similar way. For each bad element $a \in X$ and each good bucket X_u, $u \in \{0,1\}^r$, we XOR all fingerprints in the word-packed array X_u^* with $h_2(a)$ and use a lookup table to check whether it has some fingerprints in common with the word-packed array $X^*_{h_1(a) \oplus u}$ of the corresponding good bucket. If this lookup yields a positive result, we check all pairs in the corresponding buckets in time $O((\log^2 n)/(\log\log n)^2)$. As before, the expected running time is $O(R^2)$, including the expected time $O(n^2/((\log\log n)^3 \log^3 n))$ due to colliding triples.

Examining all triples with at least two bad elements can be done using a hash table as mentioned above in expected time $O(n + R^2)$.

The total expected running time for this parameter choice is $O(n^2(\log\log n)^2/\log^2 n)$. ◀

[5] Note that it would not be possible to derive expected time $O(R^2)$ for checking all pairs of bad elements if we did not start all over if the number of keys in bad buckets is at least $2R$.

Algorithm 5: The randomized subquadratic 3XOR algorithm for the case $w = \Omega((\log^2 n)\log\log n)$. For $w = o((\log^2 n)\log\log n)$ using lookup tables to search for solutions involving at most one bad element yields a faster algorithm.

Algorithm 3XOR(X): // $X \subseteq \{0,1\}^w$, $|X| = n$
  repeat
    // partition X into buckets using h_1:
    pick linear, 1-universal $h_1 : \{0,1\}^w \to \{0,1\}^r$ with $2^r = R \approx \lceil 6n(\log w)/w \rceil$
    $X_u \leftarrow \{x \in X \mid h_1(x) = u\}$ for $u \in \{0,1\}^r$
    $B \leftarrow \{x \in X \mid |X_{h(x)}| > 3\frac{n}{R}\}$ // bad elements in overfull buckets
  until $|B| < 2R$
  // search for solution involving at least two bad elements:
  for $a, b \in B$ **do** // $< 4R^2$ choices
    if $a \oplus b \in X$ **then** // $O(1)$ using appropriate hash table for X
      return $(a, b, a \oplus b)$
  // search for solution involving at most one bad element:
  $X_u \leftarrow \emptyset$ for $u \in \{0,1\}^r$ with $|X_u| > 3\frac{n}{R}$ // empty the bad buckets
  pick linear, 1-universal $h_2 : \{0,1\}^w \to \{0,1\}^p$ with $p = \lfloor 2\log w \rfloor$
  for $u \in \{0,1\}^r$ **do**
    // pack fingerprints of elements of X_u into one word X_u^*
    $X_u^* \leftarrow h_2(X_u) :=$ concatenate $\{h_2(x) \mid x \in X_u\}$
  for $a \in X$ *and* $u \in \{0,1\}^r$ **do** // $n \cdot R$ iterations
    $X_u^{*,a} \leftarrow X_u^* \oplus h_2(a)$ // $h_2(a)$ added to each fingerprint in X_u^*
    for $v \in X_u^{*,a} \cap X^*_{h_1(a)\oplus u}$ **do**
      identify responsible b, c, in particular with $v = h_2(a) \oplus h_2(b) = h_2(c)$, $h_1(b) = u$
      if $a \oplus b = c$ **then**
        return (a, b, c)
    needs time $O(\log^2(n/R))$ plus size of intersection
  return *no solution*

5 Conditional Lower Bounds from the 3XOR Conjecture

As already mentioned in Section 1, the best word RAM algorithm for int3SUM currently known [2] can solve this problem in expected time $O(n^2 \cdot \min\{\frac{\log^2 w}{w}, \frac{(\log\log n)^2}{\log^2 n}\})$ for $w = O(n \log n)$. The best deterministic algorithm [8] takes time $n^2(\log\log n)^{O(1)}/\log^2 n$. It is a popular conjecture that every algorithm for 3SUM (deterministic or randomized) needs (expected) time $n^{2-o(1)}$. Therefore, this conjectured lower bound can be used as a basis for conditional lower bounds for a wide range of other problems [15, 19, 22, 26].

Similarly, it seems natural to conjecture that every algorithm for the related 3XOR problem (deterministic or randomized) needs (expected) time $n^{2-o(1)}$. (In Theorem 7, the upper bound for short word lengths is $n^2\frac{(\log\log n)^2}{\log^2 n} = n^{2-(2\log\log n - 2\log\log\log n)/\log n}$ where $(2\log\log n - 2\log\log\log n)/\log n = o(1)$.) Therefore, it is a valid candidate for reductions to other computational problems [19, 27].

The general strategy of the subquadratic int3SUM algorithm [2], already employed in Section 4, is quite similar to the reductions in [22]. Therefore, we are able to reduce 3XOR to offline SetDisjointness and offline SetIntersection, too. Hence, the conditional lower bounds for the problems mentioned in [22] (and bounds for dynamic problems from [26]) also hold with respect to the 3XOR conjecture. A detailed discussion can be found in [27]. Below, we will outline the general proof strategy.

5.1 Offline SetDisjointness and Offline SetIntersection

We reduce 3XOR to the following two problems.

▶ **Problem 8** (Offline SetDisjointness). ***Input:*** *Finite set C, finite families A and B of subsets of C, $q \in \mathbb{N}$ pairs of subsets $(S, S') \in A \times B$.*
Task: *Find all of the q pairs (S, S') with $S \cap S' \neq \emptyset$.*

▶ **Problem 9** (Offline SetIntersection). ***Input:*** *Finite set C, finite families A and B of subsets of C, $q \in \mathbb{N}$ pairs of subsets $(S, S') \in A \times B$.*
Task: *List all elements of the intersections $S \cap S'$ of the q pairs (S, S').*

5.2 Reductions from 3XOR

By giving an expected time $\leq n^{2-\Omega(1)}$ reduction from 3XOR to offline SetDisjointness and offline SetIntersection, we can prove lower bounds for the latter two problems, conditioned on the 3XOR conjecture.

▶ **Theorem 10.** *Assume* 3XOR *requires expected time $\Omega(n^2/f(n))$ for $f(n) = n^{o(1)}$ on a word RAM. Then for $0 < \gamma < 1$ every algorithm for* offline SetDisjointness *that works on instances with $|C| = \Theta(n^{2-2\gamma})$, $|A| = |B| = \Theta(n \log n)$, $|S| = O(n^{1-\gamma})$ for all $S \in A \cup B$ and $q = \Theta(n^{1+\gamma} \log n)$ requires expected time $\Omega(n^2/f(n))$.*

▶ **Theorem 11.** *Assume* 3XOR *requires expected time $\Omega(n^2/f(n))$ for $f(n) = n^{o(1)}$ on a word RAM. Then for $0 \leq \gamma < 1$ and $\delta > 0$, every algorithm for* offline SetIntersection *which works on instances with $|C| = \Theta(n^{1+\delta-\gamma})$, $|A| = |B| = \Theta(\sqrt{n^{1+\delta+\gamma}})$, $|S| = O(n^{1-\gamma})$ for all $S \in A \cup B$, $q = \Theta(n^{1+\gamma})$ and expected output size $O(n^{2-\delta})$ requires expected time $\Omega(n^2/f(n))$.*

Proof. (For more details, see [27, ch. 6]. Algorithm 6 reduces 3XOR to offline SetDisjointness. The pseudocode implementation for offline SetIntersection is given in [11].) Let $X \subseteq \{0,1\}^w$ be the given 3XOR instance. As in Section 4, we use two levels of hashing.

At first, we hash the elements of X with a randomly chosen hash function $h_1 \in \mathcal{H}^{\text{lin}}_{w,r}$ into $R = 2^r = \Theta(n^\gamma)$ buckets in time $O(n \log n)$. Then, we apply Corollary 3: There are expected $O(R) = O(n^\gamma)$ elements in buckets with more than three times their expected size. For each such bad element, we can naively check in time $O(n \log n)$ whether it is part of a triple (a, b, c) with $a \oplus b = c$ or not. Since $\gamma < 1$, all bad elements can be checked in expected time $\leq n^{2-\Omega(1)}$. Therefore, we can assume that every bucket X_u, $u \in \{0,1\}^r$, has at most $3\frac{n}{R} = O(n^{1-\gamma})$ elements.

The second level of hashing uses two independently and randomly chosen hash functions $h_{21}, h_{22} \in \mathcal{H}^{\text{lin}}_{w,p}$ where $P = 2^{2p} = (5n/R)^2 = O(n^{2-2\gamma})$ for offline SetDisjointness and $P = 2^{2p} = n^{1+\delta}/R = O(n^{1+\delta-\gamma})$ for offline SetIntersection. (The function h_2 with $h_2(x) = h_{21}(x) \circ h_{22}(x)$, where $\circ$ denotes the concatenation of bitstrings, is randomly chosen from a linear and 1-universal class $\mathcal{H}$ of hash functions $\{0,1\}^w \to \{0,1\}^{2p}$.) The hash values can be calculated in time $O(n \log^2 n)$. (The additional $\log n$ factor is only necessary for offline SetDisjointness, since we need to use $\Theta(\log n)$ choices of hash functions h_2 to get an error probability that is small enough.) For each $u \in \{0,1\}^r$ and $v \in \{0,1\}^p$, we create "shifted" buckets $X^{\uparrow}_{u,v} = \{\, h_2(x) \oplus (v \circ 0^p) \mid x \in X_u \,\}$ and $X^{\downarrow}_{u,v} = \{\, h_2(x) \oplus (0^p \circ v) \mid x \in X_u \,\}$. One such set can be computed in time $O(n^{1-\gamma})$. Therefore, all sets can be computed in time $O(R\sqrt{P}\log n \cdot n^{1-\gamma}) = O(n^{2-\gamma}\log n)$ for offline SetDisjointness and $O(R\sqrt{P}\cdot n^{1-\gamma}) = O(n^{(3+\delta-\gamma)/2})$ for offline SetIntersection.

We can show that for all $u \in \{0,1\}^r$ and $c \in X$, if there are $a, b \in X$ such that $a \oplus b = c$ and $a \in X_u$, then $X^{\uparrow}_{u,h_{21}(c)} \cap X^{\downarrow}_{u \oplus h_1(c),h_{22}(c)} \neq \emptyset$. Therefore, we create the following offline SetDisjointness (offline SetIntersection) instance: $C := \{0,1\}^{2p}$, $A := \{\, X^{\uparrow}_{u,v} \mid u \in \{0,1\}^r, v \in \{0,1\}^p \,\}$, $B := \{\, X^{\downarrow}_{u,v} \mid u \in \{0,1\}^r, v \in \{0,1\}^p \,\}$ and q queries $(X^{\uparrow}_{u,h_{21}(c)}, X^{\downarrow}_{u \oplus h_1(c),h_{22}(c)})$ for all $u \in \{0,1\}^r$ and $c \in X$ in time $\leq n^{2-\Omega(1)}$. (These are $R \cdot n = \Theta(n^{1+\gamma})$ queries for offline SetIntersection. For offline SetDisjointness, we create $R \cdot n$ queries for each of the $\Theta(\log n)$ choices of h_2.)

After the offline SetDisjointness or offline SetIntersection instance has been solved, we can use this answer to compute the answer for X in expected time $\leq n^{2-\Omega(1)}$. We only have to check if a positive answer from offline SetDisjointness (a pair with non-empty intersection) or offline SetIntersection (an element of an intersection) yields a solution triple of X or not.

For offline SetDisjointness, we can show that the probability for a triple to yield a false positive can be made polynomially small if we consider $K = \Theta(\log n)$ choices of h_2 and only examine $(X_u \oplus c) \cap X_{h_1(c) \oplus u}$ if this is suggested by all K corresponding queries. For offline SetIntersection, the expected number of colliding triples is $O(n^{2-\delta})$. By trying to guess a good triple $\Theta(n \log n)$ times before creating the offline SetIntersection instance we can avoid a problem for the expected running time if a 3XOR instance yields an offline SetIntersection instance with output size $\omega(n^{2-\delta})$.

For all relevant values of γ and δ, the total running time is $\leq n^{2-\Omega(1)}$ in addition to the time needed to solve the offline SetDisjointness or offline SetIntersection instance. ◀

6 Conclusions and Remarks

We have presented a simple deterministic algorithm with running time $O(n^2)$. Its core is a version of the PATRICIA tree for $X \subseteq \{0,1\}^w$, which makes it possible to traverse the set $a \oplus X$ in ascending order for arbitrary $a \in \{0,1\}^w$ in linear time. Furthermore, our randomized algorithm solves the 3XOR problem in expected time $O(n^2 \cdot \min\{\frac{\log^3 w}{w}, \frac{(\log\log n)^2}{\log^2 n}\})$ for $w = O(n \log n)$, and $O(n \log^2 n)$ for $n \log n \leq w = O(2^{n \log n})$. The crossover point between the w and the $\log n$ factor is $w = (\log^2 n)\log\log n$. The only difference to the running time of [2] is in an extra factor $\log w$ in the word-length-dependent part. This is due to the necessity to re-sort a word-packed array of size $O(w/\log w)$ in time $O(\log^2 w)$ after we have XOR-ed each of its elements with a (common) element. Finally, we have reduced 3XOR to offline SetDisjointness and offline SetIntersection, establishing conditional lower bounds (as in [22] conditioned on the int3SUM conjecture).

A simple, but important observation, which is used in apparently all deterministic subquadratic time algorithms for 3SUM, is *Fredman's trick*:

$$a + b < c + d \iff a - d < c - b \qquad \text{for all } a, b, c, d \in \mathbb{Z}\,.$$

Unfortunately, such a relation does not exist in our setting, since there is no linear order $\prec$ on $\{0,1\}^w$ such that $a \oplus b \prec c \oplus d \iff a \oplus d \prec c \oplus b$ holds for all $a, b, c, d \in \{0,1\}^w$. Since all elements are self-inverse, for $a = b = c = 0^w$ and any $d \in \{0,1\}^w$, we would get $0^w \prec d \iff d \prec 0^w$. Is there another, "trivial-looking" trick for 3XOR, that establishes a basic approach to solve 3XOR in deterministic subquadratic time?

Algorithm 6: Algorithm reducing 3XOR to offline SetDisjointness, establishing a conditional lower bound on the runtime of offline SetDisjointness.

Reduction 3XOR-to-offlineSetDisjointness(X, γ)**:** // $X \subseteq \{0,1\}^w$, $|X| = n$, $0 < \gamma < 1$
  // partition X into buckets using h_1:
  pick linear, 1-universal $h_1 : \{0,1\}^w \to \{0,1\}^r$ with $2^r = R \approx \lceil n^\gamma \rceil$
  $X_u \leftarrow \{x \in X \mid h_1(x) = u\}$ for $u \in \{0,1\}^r$
  $B \leftarrow \{x \in X \mid |X_{h(x)}| > 3\frac{n}{R}\}$ // bad elements in overfull buckets
  for $b \in B$ **do** // expected $O(R)$ elements
    $X^{\oplus b} \leftarrow \mathrm{sort}\{a \oplus b \mid a \in X\}$
    if $\exists c \in X^{\oplus b} \cap X$ **then**
      return $(c \oplus b, b, c)$
  // create shifted buckets using h^i_{21}, h^i_{22}:
  pick linear, 1-universal $h^i_{21}, h^i_{22} : \{0,1\}^w \to \{0,1\}^p$ with
    $2^{2p} = P \approx \lceil (5n/R)^2 \rceil = O(n^{2-2\gamma})$ and $0 \le i < \lceil \log n \rceil$
  for $u \in \{0,1\}^r$, $v \in \{0,1\}^p$ *and* $0 \le i < \lceil \log n \rceil$ **do**
    $X^{\uparrow,i}_{u,v} \leftarrow \{(h^i_{21}(a) \oplus v, h^i_{22}(a)) \mid a \in X_u\}$
    $X^{\downarrow,i}_{u,v} \leftarrow \{(h^i_{21}(a), h^i_{22}(a) \oplus v) \mid a \in X_u\}$
  // apply algorithm for offline SetDisjointness:
  $(A, B, C, Q) \leftarrow ((X^{\uparrow,i}_{u,v})_{u,v,i}, (X^{\downarrow,i}_{u,v})_{u,v,i}, \{0,1\}^{2p}, \emptyset)$
  for $c \in X$, $u \in \{0,1\}^r$ *and* $0 \le i < \lceil \log n \rceil$ **do**
    $q \leftarrow (X^{\uparrow,i}_{u,h^i_{21}(c)}, X^{\downarrow,i}_{u \oplus h_1(c), h^i_{22}(c)})$, identified by (c, u, i)
    $Q \leftarrow Q \cup \{q\}$
  $Q' \leftarrow$ offlineSetDisjointness(A,B,C,Q) // $Q' \subseteq Q$
  // calculate solution for the 3XOR instance:
  for $c \in X$ *and* $u \in \{0,1\}^r$ **do**
    if $(c, u, i) \in Q'$ *for all* $0 \le i < \lceil \log n \rceil$ **then**
      $X^{\oplus c}_u \leftarrow \mathrm{sort}\{a \oplus c \mid a \in X_u\}$
      if $\exists b \in X^{\oplus c}_u \cap X_{h_1(c) \oplus u}$ **then**
        return $(b \oplus c, b, c)$
  return *no solution*

Another open question is how the optimal running times for 3SUM and 3XOR are related. At first sight, the two problems seem to be very similar, but the details make the difference. The observations mentioned above (especially the problem of re-sorting slightly modified word-packed arrays and the possible absence of a relation like Fredman's trick) hint at a larger gap than expected. On the other hand, the fact that both problems can be reduced to a wide variety of computational problems in a similar way (e. g. listing triangles in a graph, offline SetDisjointness and offline SetIntersection) increases hope for a more concrete dependance.

References

1 Susanne Albers and Torben Hagerup. Improved Parallel Integer Sorting without Concurrent Writing. *Inf. Comput.*, 136(1):25–51, 1997.

2 Ilya Baran, Erik D. Demaine, and Mihai Pătraşcu. Subquadratic Algorithms for 3SUM. *Algorithmica*, 50(4):584–596, 2008.

3 Ilya Baran, Erik D. Demaine, and Mihai Pătraşcu. Subquadratic Algorithms for 3SUM. In *WADS*, pages 409–421, 2005.

4 Luis Barba, Jean Cardinal, John Iacono, Stefan Langerman, Aurélien Ooms, and Noam Solomon. Subquadratic Algorithms for Algebraic Generalizations of 3SUM. In *SoCG*, volume 77 of *LIPIcs*, pages 13:1–13:15, 2017.

5 K. E. Batcher. Sorting Networks and Their Applications. In *SJCC*, AFIPS (Spring), pages 307–314, 1968.

6 Charles Bouillaguet, Claire Delaplace, and Pierre-Alain Fouque. Revisiting and Improving Algorithms for the 3XOR Problem. *IACR ToSC*, 2018(1):254–276, 2018.

7 Timothy M. Chan. The Art of Shaving Logs. WADS (Invited Talk), 2013.

8 Timothy M. Chan. More Logarithmic-Factor Speedups for 3SUM, (median, +)-Convolution, and Some Geometric 3SUM-Hard Problems. In *SODA*, pages 881–897, 2018.

9 Timothy M. Chan and Moshe Lewenstein. Clustered Integer 3SUM via Additive Combinatorics. In *STOC*, pages 31–40, 2015.

10 Martin Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *STACS*, pages 569–580, 1996.

11 Martin Dietzfelbinger, Philipp Schlag, and Stefan Walzer. A Subquadratic Algorithm for 3XOR. *CoRR*, abs/1804.11086, 2018. `arXiv:1804.11086`.

12 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *J. ACM*, 31(3):538–544, 1984.

13 Michael L. Fredman and Dan E. Willard. Surpassing the Information Theoretic Bound with Fusion Trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

14 Ari Freund. Improved Subquadratic 3SUM. *Algorithmica*, 77(2):440–458, Feb 2017.

15 Anka Gajentaan and Mark H. Overmars. On a Class of $O(n^2)$ Problems in Computational Geometry. *Comput. Geom.*, 5(3):165–185, 1995.

16 Omer Gold and Micha Sharir. Improved Bounds for 3SUM, k-SUM, and Linear Degeneracy. *CoRR*, abs/1512.05279, 2015.

17 Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic Dictionaries. *J. Algorithms*, 41(1):69–85, 2001. `doi:10.1006/jagm.2001.1171`.

18 Yijie Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. *J. Algorithms*, 50(1):96–105, 2004.

19 Zahra Jafargholi and Emanuele Viola. 3SUM, 3XOR, Triangles. *Algorithmica*, 74(1):326–343, 2016.

20 Allan Grønlund Jørgensen and Seth Pettie. Threesomes, Degenerates, and Love Triangles. In *FOCS*, pages 621–630, 2014.

21 Daniel M. Kane, Shachar Lovett, and Shay Moran. Near-optimal Linear Decision Trees for k-SUM and Related Problems. In *STOC*, pages 554–563, 2018.

22 Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3SUM conjecture. In *SODA*, pages 1272–1287, 2016.

23 Andrea Lincoln, Virginia Vassilevska Williams, Joshua R. Wang, and R. Ryan Williams. Deterministic Time-Space Tradeoffs for k-SUM. *CoRR*, abs/1605.07285, 2016.

24 Y. Mansour, N. Nisan, and P. Tiwari. The Computational Complexity of Universal Hashing. *Theor. Comput. Sci.*, 107(1):121–133, 1993.

25 Donald R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, 1968. `doi:10.1145/321479.321481`.

26 Mihai Pătraşcu. Towards Polynomial Lower Bounds for Dynamic Problems. In *STOC*, pages 603–610, 2010.

27 Philipp Schlag. Untere Schranken für Berechnungsprobleme auf der Basis der 3SUM-Vermutung. Master's thesis, TU Ilmenau, Germany, 2016.

28 Joshua R. Wang. Space-Efficient Randomized Algorithms for K-SUM. In *ESA*, pages 810–829, 2014.