Orthogonal Range Searching in Moderate Dimensions: k-d Trees and Range Trees Strike Back

Timothy M. Chan*

March 20, 2017

Abstract

We revisit the *orthogonal range searching* problem and the *exact ℓ_∞ nearest neighbor searching* problem for a static set of n points when the dimension d is moderately large. We give the first data structure with near linear space that achieves truly sublinear query time when the dimension is any constant multiple of $\log n$. Specifically, the preprocessing time and space are $O(n^{1+\delta})$ for any constant $\delta > 0$, and the expected query time is $n^{1-1/O(c \log c)}$ for $d = c \log n$. The data structure is simple and is based on a new "augmented, randomized, lopsided" variant of k-d trees. It matches (in fact, slightly improves) the performance of previous combinatorial algorithms that work only in the case of offline queries [Impagliazzo, Lovett, Paturi, and Schneider (2014) and Chan (SODA'15)]. It leads to slightly faster *combinatorial* algorithms for *all-pairs shortest paths* in general real-weighted graphs and *rectangular Boolean matrix multiplication*.

In the offline case, we show that the problem can be reduced to the *Boolean orthogonal vectors* problem and thus admits an $n^{2-1/O(\log c)}$-time non-combinatorial algorithm [Abboud, Williams, and Yu (SODA'15)]. This reduction is also simple and is based on range trees.

Finally, we use a similar approach to obtain a small improvement to Indyk's data structure [FOCS'98] for *approximate* ℓ_∞ nearest neighbor search when $d = c \log n$.

1 Introduction

In this paper, we revisit some classical problems in computational geometry:

- In *orthogonal range searching*, we want to preprocess n data points in $\mathbb{R}^d$ so that we can detect if there is a data point inside any query axis-aligned box, or report or count all such points.
- In *dominance range searching*, we are interested in the special case when the query box is d-sided, of the form $(-\infty, q_1] \times \cdots \times (-\infty, q_d]$; in other words, we want to detect if there is a data point $(p_1, \ldots, p_d)$ that is dominated by a query point $(q_1, \ldots, q_d)$, in the sense that $p_j \leq q_j$ for all $j \in \{1, \ldots, d\}$, or report or count all such points.
- In *ℓ_∞ nearest neighbor searching*, we want to preprocess n data points in $\mathbb{R}^d$ so that we can find the nearest neighbor to the given query point under the ℓ_∞ metric.

*Department of Computer Science, University of Illinois at Urbana-Champaign (tmc@illinois.edu). This work was done while the author was at the Cheriton School of Computer Science, University of Waterloo.

All three problems are related. Orthogonal range searching in d dimensions reduces to dominance range searching in $2d$ dimensions.[1] Furthermore, ignoring logarithmic factors, ℓ_∞ nearest neighbor searching reduces to its decision problem (deciding whether the ℓ_∞ nearest neighbor distance to a given query point is at most a given radius) by parametric search or randomized search [8], and the decision problem clearly reduces to orthogonal range searching.

The standard *k-d tree* [23] has $O(dn \log n)$ preprocessing time and $O(dn)$ space, but the worst-case query time is $O(dn^{1-1/d})$. The standard *range tree* [23] requires $O(n \log^d n)$ preprocessing time and space and $O(\log^d n)$ query time, excluding an $O(K)$ term for the reporting version of the problem with output size K. Much work in computational geometry has been devoted to small improvements of a few logarithmic factors. For example, the current best result for orthogonal range reporting has $O(n \log^{d-3+\varepsilon} n)$ space and $O(\log^{d-3} n / \log^{d-4} \log n + K)$ time [12]; there are also other small improvements for various offline versions of the problems [12, 13, 2].

In this paper, we are concerned with the setting *when the dimension is nonconstant.* Traditional approaches from computational geometry tend to suffer from exponential dependencies in d (the so-called "curse of dimensionality"). For example, the $O(dn^{1-1/d})$ or $O(\log^d n)$ query time bound for range trees or k-d trees is sublinear only when $d \ll \log n / \log\log n$. By a more careful analysis [9], one can show that range trees still have sublinear query time when $d \ll \alpha_0 \log n$ for a sufficiently small constant α_0. The case when the dimension is close to logarithmic in n is interesting in view of known dimensionality reduction techniques [17] (although such techniques technically are not applicable to exact problems and, even with approximation, do not work well for ℓ_∞). The case of polylogarithmic dimensions is also useful in certain *non-geometric* applications such as all-pairs shortest paths (as we explain later). From a theoretical perspective, it is important to understand when the time complexity transitions from sublinear to superlinear.

Previous offline results. We first consider the *offline* version of the problems where we want to answer a batch of n queries all given in advance. In high dimensions, it is possible to do better than $O(dn^2)$-time brute-force search, by a method of Matoušek [22] using fast (rectangular) matrix multiplication [21]; for example, we can get $n^{2+o(1)}$ time for $d \ll n^{0.15}$. However, this approach inherently cannot give *sub*quadratic bounds.

In 2014, a surprising discovery was made by Impagliazzo et al. [18]: range-tree-like divide-and-conquer can still work well even when the dimension goes a bit above logarithmic. Their algorithm can answer n offline dominance range queries (and thus orthogonal range queries and ℓ_∞ nearest neighbor queries) in total time $n^{2-1/O(c^{15} \log c)}$ (ignoring an $O(K)$ term for reporting) in dimension $d = c \log n$ for any possibly nonconstant c ranging from 1 to about $\log^{1/15} n$ (ignoring $\log\log n$ factors). Shortly after, by a more careful analysis of the same algorithm, Chan [11] refined the time bound to $n^{2-1/O(c \log^2 c)}$, which is subquadratic for c up to about $\log n$, i.e., dimension up to about $\log^2 n$.

At SODA'15, Abboud, Williams, and Yu [1] obtained an even better time bound for dominance range detection in the *Boolean* special case, where all coordinate values are 0's and 1's (in this case, the problem is better known as the *Boolean orthogonal vectors* problem[2]). The total time for n offline Boolean dominance range detection queries is $n^{2-1/O(\log c)}$. The bound $n^{2-1/O(\log c)}$ is a

[1] $(p_1, \ldots, p_d)$ is inside the box $[a_1, b_1] \times \cdots \times [a_d, b_d]$ iff $(-p_1, p_1, \ldots, -p_d, p_d)$ is dominated by $(-a_1, b_1, \ldots, -a_d, b_d)$ in $\mathbb{R}^{2d}$.

[2] Two vectors $(p_1, \ldots, p_d), (q_1, \ldots, q_d) \in \{0,1\}^d$ are orthogonal iff $\sum_{i=1}^{d} p_i q_i = 0$ iff $(p_1, \ldots, p_d)$ is dominated by $(1 - q_1, \ldots, 1 - q_d)$ (recalling that our definition of dominance uses non-strict inequality).

natural barrier, since a faster offline Boolean dominance algorithm would imply an algorithm for CNF-SAT with n variables and cn clauses that would beat the currently known $2^{n(1-1/O(\log c))}$ time bound [1]; and an $O(n^{2-\delta})$-time algorithm for any $c = \omega(1)$ would break the strong exponential-time hypothesis (SETH) [25]. Abboud et al.'s algorithm was based on the *polynomial method* pioneered by Williams [27] (see [5, 4] for other geometric applications). The algorithm was originally randomized but was subsequently derandomized by Chan and Williams [14] in SODA'16 (who also extended the result from detection to counting).

Abboud et al.'s approach has two main drawbacks, besides being applicable to the Boolean case only: 1. it is not "combinatorial" and relies on fast rectangular matrix multiplication, making the approach less likely to be practical, and 2. it only works in the offline setting.

Impagliazzo et al.'s range-tree method [18] is also inherently restricted to the offline setting—in their method, the choice of dividing hyerplanes crucially requires knowledge of all query points in advance. All this raises an intriguing open question: are there nontrivial results for online queries in $d = c \log n$ dimensions?

New online result. In Section 2.1, we resolve this question by presenting a randomized data structure with $O(n^{1+\delta})$ preprocessing time and space that can answer online dominance range queries (and thus orthogonal range queries and ℓ_∞ nearest neighbor queries) in $n^{1-1/O(c \log^2 c)}$ expected time for any $d = c \log n \ll \log^2 n / \log\log n$ and for any constant $\delta > 0$. (We assume an oblivious adversary, i.e., that query points are independent of the random choices made by the preprocessing algorithm.) The total time for n queries is $n^{2-1/O(c \log^2 c)}$, matching the offline bound from Impagliazzo et al. [18] and Chan [11]. The method is purely combinatorial, i.e., does not rely on fast matrix multiplication.

More remarkable than the result perhaps is the simplicity of the solution: it is just a variant of k-d trees! More specifically, the dividing hyperplane is chosen in a "lopsided" manner, along a randomly chosen coordinate axis; each node is augmented with secondary structures for some lower-dimensional projections of the data points. The result is surprising, considering the longstanding popularity of k-d trees among practitioners. Our contribution lies in recognizing, and proving, that they can have good theoretical worst-case performance. (Simple algorithms with nonobvious analyses are arguably the best kind.)

In Section 2.2, we also describe a small improvement of the query time to $n^{1-1/O(c \log c)}$. This involves an interesting application of so-called *covering designs* (from combinatorics), not often seen in computational geometry.

Applications. By combining with previous techniques [9, 11], our method leads to new results for two classical, non-geometric problems: *all-pairs shortest paths (APSP)* and *Boolean matrix multiplication (BMM)*.

- We obtain a new combinatorial algorithm for solving the APSP problem for arbitrary real-weighted graphs with n vertices (or equivalently the *(min,+) matrix multiplication* problem for two $n \times n$ real-valued matrices) in $O((n^3/\log^3 n)\,\mathrm{poly}(\log\log n))$ time; see Section 2.4. This is about a logarithmic factor faster than the best previous combinatorial algorithm [10, 16, 11], not relying on fast matrix multiplication à la Strassen. It also extends Chan's combinatorial algorithm for Boolean matrix multiplication from SODA'15 [11], which has a similar running time (although for Boolean matrix multiplication, Yu [28] has recently obtained a further logarithmic-factor improvement).

This extension is intriguing, as (min,+) matrix multiplication over the reals appears tougher than other problems such as standard matrix multiplication over $\mathbb{F}_2$, for which the well-known "four Russians" time bound of $O(n^3/\log^2 n)$ [7] has still not been improved for combinatorial algorithms.

- We obtain a new combinatorial algorithm to multiply an $n \times \log^2 n$ and a $\log^2 n \times n$ Boolean matrix in $O((n^2/\log n)\,\text{poly}(\log\log n))$ time, which is almost optimal in the standard word RAM model since the output requires $\Omega(n^2/\log n)$ words; see Section 2.5. The previous combinatorial algorithm by Chan [11] can multiply an $n \times \log^3 n$ and a $\log^3 n \times n$ Boolean matrix in $O(n^2\,\text{poly}(\log\log n))$ time. The new result implies the old, but not vice versa.

New offline result. Returning to the offline dominance or orthogonal range searching problem, Abboud, Williams, and Yu's non-combinatorial algorithm [1] has a better $n^{2-1/O(\log c)}$ time bound but is only for the Boolean case, leading to researchers to ask whether the same result holds for the more general problem for real input. In one section of Chan and Williams' paper [14], such a result was obtained but only for $d \approx 2^{\Theta(\sqrt{\log n})}$.

In Section 3, we resolve this question by giving a black-box reduction from the real case to the Boolean case, in particular, yielding $n^{2-1/O(\log c)}$ time for any $d = c\log n \ll 2^{\Theta(\sqrt{\log n})}$.

This equivalence between general dominance searching and the Boolean orthogonal vectors problem is noteworthy, since the Boolean orthogonal vectors problem has been used as a basis for numerous conditional hardness results in P.

As one immediate application, we can now solve the *integer linear programming* problem on n variables and cn constraints in $2^{(1-1/O(\log c))n}$ time, improving Impagliazzo et al.'s $2^{(1-1/\text{poly}(c))n}$ algorithm [18].

Our new reduction is simple, this time, using a range-tree-like recursion.

Approximate ℓ_∞ nearest neighbor searching. So far, our discussion has been focused on exact algorithms. We now turn to ℓ_∞ nearest neighbor searching in the *approximate* setting. By known reductions (ignoring polylogarithmic factors) [17], it suffices to consider the fixed-radius decision problem: deciding whether the nearest neighbor distance is approximately less than a fixed value. Indyk [19] provided the best data structure for the problem, achieving $O(\log_\rho \log d)$ approximation factor, $O(dn^\rho \log n)$ preprocessing time, $O(dn^\rho)$ space, and $O(d\log n)$ query time for any ρ ranging from 1 to $\log d$. The data structure is actually based on traditional-style geometric divide-and-conquer. Andoni, Croitoru, and Pătraşcu [6] proved a nearly matching lower bound.

In Section 4.1, we improve the approximation factor of Indyk's data structure to $O(\log_\rho \log c)$ for dimension $d = c\log n$, for any ρ ranging from $1+\delta$ to $\log c$ (as an unintended byproduct, we also improve Indyk's query time to $O(d)$). The improvement in the approximation factor is noticeable when the dimension is close to logarithmic. It does not contradict Andoni et al.'s lower bound [6], since their proof assumed $d \gg \log^{1+\Omega(1)} n$.

For example, by setting $\rho \approx \log c$, we get $O(1)$ approximation factor, $n^{O(\log c)}$ preprocessing time/space, and $O(d)$ query time. By dividing into $n^{1-\alpha}$ groups of size n^α, we can lower the preprocessing time/space to $n^{1-\alpha} \cdot (n^\alpha)^{O(\log(c/\alpha))}$ while increasing the query time to $O(dn^{1-\alpha})$. Setting $\alpha \approx 1/\log c$, we can thus answer n (online) queries with $O(1)$ approximation factor in $n^{2-1/O(\log c)}$ total time, which curiously matches our earlier result for exact ℓ_∞ nearest neighbor search but by a purely combinatorial algorithm.

In Section 4.2, we also provide an alternative data structure with linear space but a larger $O(c^{(1-\rho)/\rho^2})$ approximation factor, and $O(dn^{\rho+\delta})$ query time for any ρ between δ and $1-\delta$.

The idea is to modify Indyk's method to incorporate, once again, a range-tree-like recursion.

2 Online Dominance Range Searching

In this section, we study data structures for online orthogonal range searching in the reporting version (counting or detection can be dealt with similarly), using only combinatorial techniques without fast matrix multiplication. By doubling the dimension (footnote 1), it suffices to consider the dominance case.

2.1 Main Data Structure

Our data structure is an augmented, randomized lopsided variant of the k-d tree, where each node contains secondary structures for various lower-dimensional projections of the input.

Data structure. Let $\delta \in (0,1)$ and $c \in [\delta C_0, (\delta/C_0)\log N/\log^2\log N]$ be user-specified parameters, for a sufficiently large constant C_0, where N is a fixed upper bound on the size of the input point set. Let $b \geq 2$ and $\alpha \in (0, 1/2)$ be parameters to be chosen later.

Given a set P of $n \leq N$ data points in $d \leq c\log N$ dimensions, our data structure is simple and is constructed as follows:

0. If $n \leq 1/\alpha$ or $d = 0$, then just store the given points.

1. Otherwise, let $\mathcal{J}$ be the collection of all subsets of $\{1, \ldots, d\}$ of size $\lfloor d/b \rfloor$. Then $|\mathcal{J}| = \binom{d}{\lfloor d/b \rfloor} = b^{O(d/b)}$. For each $J \in \mathcal{J}$, recursively[3] construct a data structure for the projection P_J of P that keeps only the coordinate positions in J.

2. Pick a random $i^* \in \{1, \ldots, d\}$. Let $\mu(i^*)$ be the $\lceil(1-\alpha)n\rceil$-th smallest i^*-th coordinate value in P; let $p(i^*)$ be the corresponding point in P. Store n, i^*, and $p(i^*)$. Recursively construct data structures for
   - the subset P_L of all points in P with i^*-th coordinate less than $\mu(i^*)$, and
   - the subset P_R of all points in P with i^*-th coordinate greater than $\mu(i^*)$.

Analysis. The preprocessing time and space satisfy the recurrence

$$T_d(n) \;\leq\; T_d(\lfloor \alpha n \rfloor) + T_d(\lfloor (1-\alpha)n \rfloor) + b^{O(d/b)} T_{\lfloor d/b \rfloor}(n) + O(n),$$

with $T_d(n) = O(n)$ for the base case $n \leq 1/\alpha$ or $d = 0$. This solves to

$$\begin{aligned} T_d(N) &\leq b^{O(d/b + d/b^2 + \cdots)} N (\log_{1/(1-\alpha)} N)^{O(\log_b d)} \\ &= b^{O(d/b)} N ((1/\alpha)\log N)^{O(\log_b d)} \\ &= N^{1+O((c/b)\log b)} 2^{O(\log((1/\alpha)\log N)\log_b d)} \;\leq\; N^{1+O(\delta)} 2^{O(\log^2((1/\alpha)\log N))} \end{aligned}$$

by setting $b := (c/\delta)\log(c/\delta)$.

[3]There are other options beside recursion here; for example, we could just use a range tree for P_J.

Query algorithm. Given the preprocessed set P and a query point $q = (q_1, \ldots, q_d)$, our query algorithm proceeds as follows.

0. If $n \le 1/\alpha$ or $d = 0$, then answer the query directly by brute-force search.
1. Otherwise, let $J_q = \{i \in \{1, \ldots, d\} : q_i \ne \infty\}$. If $|J_q| \le d/b$, then recursively answer the query for P_{J_q} and the projection of q with respect to J_q.
2. Else,
   - if $q_{i^*} \le \mu(i^*)$, then recursively answer the query for P_L and q;
   - if $q_{i^*} > \mu(i^*)$, then recursively answer the query for P_R and q, and recursively answer the query for P_L and $q' = (q_1, \ldots, q_{i^*-1}, \infty, q_{i^*+1}, \ldots, q_d)$;
   - in addition, if q dominates $p(i^*)$, then output $p(i^*)$.

Analysis. We assume that the query point q is independent of the random choices made during the preprocessing of P. Let $L_q = \{i \in \{1, \ldots, d\} : \mu(i) < q_i \ne \infty\}$. Let $j = |J_q|$ and $\ell = |L_q|$.

Suppose that $j > d/b$. The probability that we make a recursive call for P_R is equal to $\Pr[(i^* \in L_q) \vee (i^* \notin J_q)] = \ell/d + (1 - j/d)$. We always make a recursive call for P_L, either for q or a point q' with $j - 1$ non-∞ values; the probability of the latter is equal to $\Pr[i^* \in L_q] = \ell/d$.

Hence, the expected number of leaves in the recursion satisfies the following recurrence:

$$Q_{d,j}(n) \;\le\; \begin{cases} Q_{\lfloor d/b \rfloor, j}(n) & \text{if } j \le d/b \\ \max_{\ell \le j} \left[\left(\frac{\ell}{d} + 1 - \frac{j}{d}\right) Q_{d,j}(\lfloor \alpha n \rfloor) + \left(\frac{\ell}{d}\right) Q_{d,j-1}(\lfloor (1-\alpha) n \rfloor) \right. & \\ \qquad \left. + \left(1 - \frac{\ell}{d}\right) Q_{d,j}(\lfloor (1-\alpha) n \rfloor) \right] & \text{if } j > d/b, \end{cases} \tag{1}$$

with $Q_{d,j}(n) = 1$ for the base case $n \le 1/\alpha$ or $d = 0$.

This recurrence looks complicated. Following [11], one way to solve it is by "guessing". We guess that

$$Q_{d,j}(n) \le (1+\gamma)^j n^{1-\varepsilon}$$

for some choice of parameters $\gamma, \varepsilon \in (0, 1/2)$ to be specified later. We verify the guess by induction.

The base case $n \le 1/\alpha$ or $d = 0$ is trivial. Assume that the guess is true for lexicographically smaller tuples (d, j, n). For $j \le d/b$, the induction trivially goes through. So assume $j > d/b$. Let ℓ be the index that attains the maximum in (1). Then

$$\begin{aligned} Q_{d,j}(n) \;&\le\; \left(\frac{\ell}{d} + 1 - \frac{j}{d}\right)(1+\gamma)^j (\alpha n)^{1-\varepsilon} + \left(\frac{\ell}{d}\right)(1+\gamma)^{j-1}((1-\alpha)n)^{1-\varepsilon} + \\ &\quad\; \left(1 - \frac{\ell}{d}\right)(1+\gamma)^j((1-\alpha)n)^{1-\varepsilon} \\ &=\; \left[\left(\frac{\ell}{d} + 1 - \frac{j}{d}\right)\alpha^{1-\varepsilon} + \left(\frac{\ell}{d} \cdot \frac{1}{1+\gamma} + 1 - \frac{\ell}{d}\right)(1-\alpha)^{1-\varepsilon}\right](1+\gamma)^j n^{1-\varepsilon} \\ &\le\; \left[\left(1 - \frac{j-\ell}{d}\right)\alpha^{1-\varepsilon} + \left(1 - \frac{\gamma\ell}{2d}\right)(1-\alpha)^{1-\varepsilon}\right](1+\gamma)^j n^{1-\varepsilon} \\ &\le\; (1+\gamma)^j n^{1-\varepsilon}. \end{aligned}$$

For the last inequality, we need to upper-bound the following expression by 1:

$$\left(1-\frac{j-\ell}{d}\right)\alpha^{1-\varepsilon}+\left(1-\frac{\gamma\ell}{2d}\right)(1-\alpha)^{1-\varepsilon}. \tag{2}$$

- Case I: $j-\ell > d/(2b)$. Then (2) is at most

$$\begin{aligned}\left(1-\frac{1}{2b}\right)\alpha^{1-\varepsilon}+(1-\alpha)^{1-\varepsilon} &\le \left(1-\frac{1}{2b}\right)\alpha e^{\varepsilon\ln(1/\alpha)}+1-(1-\varepsilon)\alpha \\ &\le \left(1-\frac{1}{2b}\right)\alpha(1+2\varepsilon\log(1/\alpha))+1-(1-\varepsilon)\alpha \\ &\le 1-\frac{\alpha}{2b}+3\alpha\varepsilon\log(1/\alpha),\end{aligned}$$

  which is indeed at most 1 by setting $\varepsilon := 1/(6b\log(1/\alpha))$.

- Case II: $\ell > d/(2b)$. Then (2) is at most

$$\begin{aligned}\alpha^{1-\varepsilon}+1-\frac{\gamma}{4b} &\le \alpha e^{\varepsilon\ln(1/\alpha)}+1-\frac{\gamma}{4b} \\ &\le \alpha(1+2\varepsilon\log(1/\alpha))+1-\frac{\gamma}{4b} \\ &\le 2\alpha+1-\frac{\gamma}{4b},\end{aligned}$$

  which is indeed at most 1 by setting $\gamma := 8b\alpha$.

We can set $\alpha := 1/b^4$, for example. Then $\gamma = O(1/b^3)$. We conclude that

$$Q_d(N) \;\le\; (1+\gamma)^d N^{1-\varepsilon} \;\le\; e^{\gamma d}N^{1-\varepsilon} \;\le\; N^{1-\varepsilon+O(c\gamma)} \;\le\; N^{1-1/O(b\log b)}.$$

Now, $Q_d(N)$ only counts the number of leaves in the recursion. The recursion has depth $O(\log_{1/(1-\alpha)} N + \log d)$. Each internal node of the recursion has cost $O(d)$, and each leaf has cost $O(d/\alpha)$, excluding the cost of outputting points (which occurs during the base case $d=0$). Thus, the actual expected query time can be bounded by $Q_d(N)(bd\log N)^{O(1)}$, which is $N^{1-1/O(b\log b)}$ for $b \ll \log N/\log^2\log N$. As $b=(c/\delta)\log(c/\delta)$, the bound is $N^{1-1/O((c/\delta)\log^2(c/\delta))}$.

2.2 Slightly Improved Version

We now describe a small improvement to the data structure in Section 2.1, removing one $\log(c/\delta)$ factor from the exponent of the query time.

The idea is to replace $\mathcal{J}$ with a collection of slightly larger subsets, but with fewer subsets, so that any set J_q of size $t := \lfloor d/b\rfloor$ is covered by some subset in $J\in\mathcal{J}$. Such a collection is called a *covering design* (e.g., see [15]), which can be constructed easily by random sampling, as explained in part (i) of the lemma below. In our application, we also need a good time bound for finding such a $J\in\mathcal{J}$ for a given query set J_q; this is addressed in part (ii) of the lemma.

Lemma 2.1. (Covering designs) *Given numbers $v \geq k \geq t$ and N, and given a size-v ground set V,*

(i) *we can construct a collection $\mathcal{J}$ of at most $\left(\binom{v}{t}/\binom{k}{t}\right)\ln N$ size-k subsets of V in $O(v|\mathcal{J}|)$ time, so that given any query size-t subset $J_q \subset V$, we can find a subset $J \in \mathcal{J}$ containing J_q in $O(v|\mathcal{J}|)$ time with success probability at least $1 - 1/N$;*

(ii) *alternatively, with a larger collection $\mathcal{J}$ of at most $\left(\binom{v}{t}/\binom{k}{t}\right)^2 \ln^2(vN)$ subsets, we can reduce the query time to $O(v^3 \log^2(vN))$.*

Proof. Part (i) is simple: just pick a collection $\mathcal{J}$ of $\left(\binom{v}{t}/\binom{k}{t}\right)\ln N$ random size-k subsets of V. Given a query size-t subset J_q, use brute-force search. The probability that J_q is contained in a random size-k subset is $p := \binom{v-t}{k-t}/\binom{v}{k} = \binom{k}{t}/\binom{v}{t}$. Thus, the probability that J_q is not contained in any of the $|\mathcal{J}|$ random subsets is at most $(1-p)^{|\mathcal{J}|} \leq e^{-p|\mathcal{J}|} = 1/N$.

For part (ii), we use a recursive construction. Pick the largest $v' \in (k, v)$ such that $\binom{v}{t}/\binom{v'}{t} \geq \ln N$; if no such v' exists, set $v' = k$. Apply part (i) to obtain a collection $\mathcal{J}'$ of $\left(\binom{v}{t}/\binom{v'}{t}\right)\ln N$ size-v' subsets of V. For each $V' \in \mathcal{J}'$, recursively generate a collection with V' as the ground set. We have $|\mathcal{J}'| \leq \left(\binom{v}{t}/\binom{v'}{t}\right)^2$ if $v' > k$. Thus, the total number of sets in the collection satisfies the recurrence

$$C(v,k,t) \leq \left(\binom{v}{t}\Big/\binom{v'}{t}\right)^2 C(v',k,t)$$

if $v' > k$, and $C(v,k,t) \leq \ln^2 N$ otherwise. Expanding the recurrence yields a telescoping product, implying $C(v,k,t) \leq \left(\binom{v}{t}/\binom{k}{t}\right)^2 \ln^2 N$.

To answer a query for a subset $J_q \subset V$ of size t, first find a $V' \in \mathcal{J}'$ that contains Q and then recursively answer the query in V'. Since maximality of v' implies $\binom{v}{t}/\binom{v'+1}{t} < \ln N$, we have $|\mathcal{J}'| = \left(\binom{v}{t}/\binom{v'}{t}\right)\ln N < v \ln^2 N$. Thus, the query time satisfies the recurrence

$$Q(v,k,t) \leq O(v^2 \log^2 N) + Q(v',k,t),$$

which solves to $Q(v,k,t) \leq O(v^3 \log^2 N)$. The overall failure probability for a query is at most v/N, which can be changed to $1/N$ by readjusting N by a factor of v. $\square$

We now modify the data structure in Section 2.1 as follows. In step 1, we change $\mathcal{J}$ to a collection of size-$\lfloor d/2 \rfloor$ subsets of $\{1,\ldots,d\}$ obtained from Lemma 2.1(ii) with $(v,k,t) = (d, \lfloor d/2 \rfloor, \lfloor d/b \rfloor)$. Then $|\mathcal{J}| \leq \left(\binom{d}{\lfloor d/b \rfloor}/\binom{\lfloor d/2 \rfloor}{\lfloor d/b \rfloor}\right)^2 \ln^2(dN) \leq 2^{O(d/b)} \log^2 N$. The recurrence for the preprocessing time and space then improves to

$$T_d(n) \;\leq\; T_d(\lfloor \alpha n \rfloor) + T_d(\lfloor (1-\alpha)n \rfloor) + (2^{O(d/b)} \log^2 N) T_{\lfloor d/b \rfloor}(n) + O(n),$$

which solves to $T_d(N) \leq 2^{O(d/b + d/b^2 + \cdots)} N (\log_{1/(1-\alpha)} N)^{O(\log_b d)} \leq N^{1+O(\delta)} 2^{O(\log^2((1/\alpha)\log N))}$, this time by setting $b := c/\delta$ (instead of $b := (c/\delta)\log(c/\delta)$).

In the query algorithm, we modify step 1 by finding a set $J \in \mathcal{J}$ containing J_q by Lemma 2.1(ii) and recursively querying P_J (instead of P_{J_q}). If no such J exists, we can afford to switch to brute-force search, since this happens with probability less than $1/N$. The analysis of the recurrence for $Q_d(N)$ remains the same. Each internal node of the recursion now has cost $O(d^3 \log^2 N)$ by Lemma 2.1(ii); the extra factor will not affect the final bound. The overall query time is still $N^{1-1/O(b \log b)}$, which is now $N^{1-1/O((c/\delta)\log(c/\delta))}$.

Theorem 2.2. *Let $\delta > 0$ be any fixed constant and $c \in [C_1, (1/C_1) \log N / \log^2 \log N]$ for a sufficiently large constant C_1. Given N points in $d = c \log N$ dimensions, we can construct a data structure in $O(N^{1+\delta})$ preprocessing time and space, so that for any query point, we can answer a dominance range reporting query in $N^{1-1/O(c \log c)} + O(K)$ expected time where K is the number of reported points. For dominance range counting, we get the same time bound but without the K term.*

We mention one application to online (min,+) matrix-vector multiplication. The corollary below follows immediately from a simple reduction [9] to d instances of d-dimensional dominance range reporting with disjoint output.[4]

Corollary 2.3. *Let $\delta > 0$ be any fixed constant and $d = (1/C_1) \log^2 N / \log^2 \log N$ for a sufficiently large constant C_1. We can preprocess an $N \times d$ real-valued matrix A in $O(N^{1+\delta})$ time, so that given a query real-valued d-dimensional vector x, we can compute the (min,+)-product of A and x in $O(N)$ expected time.*

Applying the above corollary N/d times yields:

Corollary 2.4. *Let $\delta > 0$ be any fixed constant. We can preprocess an $N \times N$ real-valued matrix A in $O(N^{2+\delta})$ time, so that given a query N-dimensional real-valued vector x, we can compute the (min,+)-product of A and x in $O((N^2/\log^2 N) \log^2 \log N)$ expected time.*

A similar result was obtained by Williams [26] for online *Boolean* matrix-vector multiplication. Recently Larsen and Williams [20] have found a faster algorithm, in the Boolean case, but it is not combinatorial, requires amortization, and does not deal with the rectangular matrix case in Corollary 2.3.

2.3 Offline Deterministic Version

In this subsection, we sketch how to derandomize the algorithm in Section 2.1, with the improvement from Appendix 2.2, in the offline setting when all the query points are known in advance. The derandomization is achieved by standard techniques, namely, the method of conditional expectation.

We first derandomize Lemma 2.1(i) in the offline setting when the collection $\mathcal{Q}$ of all query subsets J_q is given in advance. We know that $\mathbb{E}_J[|\{J_q \in \mathcal{Q} : J_q \subseteq J\}|] = p|\mathcal{Q}|$ with $p := \binom{k}{t}/\binom{v}{t}$, over a random size-$k$ subset J of V. We explicitly find a size-k subset J such that $|\{J_q \in \mathcal{Q} : J_q \subseteq J\}|$ is at least the expected value as follows. Say $V = \{1, \ldots, v\}$. Suppose at the beginning of the i-th iteration, we have determined a set $J_{i-1} \subseteq \{1, \ldots, i-1\}$. We compute the conditional expectations $\mathbb{E}_J[|\{J_q \in \mathcal{Q} : J_q \subseteq J\}| \mid J \cap \{1, \ldots, i\} = J_{i-1}]$ and $\mathbb{E}_J[|\{J_q \in \mathcal{Q} : J_q \subseteq J\}| \mid J \cap \{1, \ldots, i\} = J_{i-1} \cup \{i\}]$. The expectations are easy to compute in $O(v|\mathcal{Q}|)$ time. If the former is larger, set $J_i = J_{i-1}$, else set $J_i = J_{i-1} \cup \{i\}$. Then J_v has the desired property, and the total for the v iterations is $O(v^2|\mathcal{Q}|)$.

[4]For any $j_0 \in \{1, \ldots, d\}$, the key observation is that $\min_{j=1}^{d}(a_{ij} + x_j) = a_{ij_0} + x_{j_0}$ iff $(a_{ij_0} - a_{i1}, \ldots, a_{ij_0} - a_{id})$ is dominated by $(x_1 - x_{j_0}, \ldots, x_d - x_{j_0})$ in $\mathbb{R}^d$.

Once this subset J is found, we can add J to the collection $\mathcal{J}$, remove all $J_q \in \mathcal{Q}$ contained in J, and repeat. Since each round removes $p|\mathcal{Q}|$ subsets from $\mathcal{Q}$, we have $|\mathcal{J}| = O(\log_{1/(1-p)} |\mathcal{Q}|) = \left(\binom{v}{t}/\binom{k}{t}\right) \cdot O(\log N)$ for $|\mathcal{Q}| \leq N$. The total time is $O(v^2|\mathcal{J}||\mathcal{Q}|)$, i.e., the amortized time per query is $O(v^2|\mathcal{J}|)$. (The extra $v^{O(1)}$ factor will not affect the final bound.) This collection $\mathcal{J}$ guarantees success for all query subsets $J_q \in \mathcal{Q}$.

The derandomization of Lemma 2.1(ii) follows from that of Lemma 2.1(i).

It remains to derandomize the preprocessing algorithm in Section 2.1. For a set Q of query points, define the cost function

$$f(Q, n) := \sum_{q \in Q} (1+\gamma)^{|J_q|} n^{1-\varepsilon}.$$

Let $Q_L(i)$ and $Q_R(i)$ be the set of query points passed to P_L and P_R by our query algorithm when i^* is chosen to be i. From our analysis, we know that

$$\mathbb{E}_{i^*}\left[f(Q_L(i^*), (1-\alpha)n) + f(Q_R(i^*), \alpha n)\right] \;\leq\; f(Q, n).$$

We explicitly pick an $i^* \in \{1, \ldots, d\}$ that minimizes $f(Q_L(i^*), (1-\alpha)n) + f(Q_R(i^*), \alpha n)$. The cost function f is easy to evaluate in $O(d|Q|)$ time, and so we can find i^* in $O(d^2|Q|)$ time. The amortized time per query increases by an extra $d^{O(1)}$ factor (which will not affect the final bound).

2.4 Offline Packed-Output Version, with Application to APSP

In this subsection, we discuss how to refine the algorithm in Section 2.1, so that the output can be reported in roughly $O(K/\log n)$ time instead of $O(K)$ in the offline setting. The approach is to combine the algorithm with bit-packing tricks.

We assume a w-bit word RAM model which allows for certain exotic word operations. In the case of $w := \delta_0 \log N$ for a sufficiently small constant $\delta_0 > 0$, exotic operations can be simulated in constant time by table lookup; the precomputation of the tables requires only $N^{O(\delta_0)}$ time.

We begin with techniques to represent and manipulate sparse sets of integers in the word RAM model. Let z be a parameter to be set later. In what follows, an interval $[a, b)$ refers to the integer set $\{a, a+1, \ldots, b-1\}$. A *block* refers to an interval of the form $[kz, (k+1)z)$. Given a set S of integers over an interval I of length n, we define its *compressed representation* to be a doubly linked list of *mini-sets*, where for each of the $O(\lceil n/z \rceil)$ blocks B intersecting I (in sorted order), we store the *mini-set* $\{j \bmod z : j \in S \cap B\}$, which consists of small $(\log z)$-bit numbers and can be packed in $O((|S \cap B|/w) \log z + 1)$ words. The total number of words in the compressed representation is $O((|S|/w) \log z + n/z + 1)$.

Lemma 2.5. (Bit-packing tricks)

(i) *Given compressed representations of two sets S_1 and S_2 over two disjoint intervals, we can compute the compressed representation of $S_1 \cup S_2$ in $O(1)$ time.*

(ii) *Given compressed representations of $S_0, \ldots, S_{m-1} \subset [0, n)$, we can compute the compressed representations of $T_0, \ldots, T_{n-1} \subset [0, m)$ with $T_j = \{i : j \in S_i\}$ (called the* transposition *of $S_0, \ldots, S_{m-1}$), in $O((K/w) \log^2 z + mn/z + m + n + z)$ time, where $K = \sum_{i=0}^{m-1} |S_i|$.*

(iii) *Given compressed representations of $S_0, \ldots, S_{m-1} \subset [0, n)$ and a bijective function $\pi : [0, n) \to [0, n)$ which is evaluable in constant time, we can compute compressed representations of $\pi(S_1), \ldots, \pi(S_m)$ in $O((K/w)\log^2 z + mn/z + m + n + z)$ time, where $K = \sum_{i=0}^{m-1} |S_i|$.*

Proof. Part (i) is straightforward by concatenating two doubly linked lists and unioning two mini-sets (which requires fixing $O(1)$ words).

For part (ii), fix a block B_1 intersecting $[0, m)$ and a block B_2 intersecting $[0, n)$. From the mini-sets $\{j \bmod z : j \in S_i \cap B_2\}$ over all $i \in B_1$, construct the list $L := \{(i \bmod z, j \bmod z) : j \in S_i,\ i \in B_1,\ j \in B_2\}$ in $O((|L|/w)\log z + z)$ time. Sort L by the second coordinate; this takes $O((|L|/w)\log^2 z)$ time by a packed-word variant of mergesort [3]. By scanning the sorted list L, we can then extract the transposed small sets $\{i \bmod z : i \in T_j \cap B_1\}$ for all $j \in B_2$. The total time over all $\lceil m/z \rceil \cdot \lceil n/z \rceil$ pairs of blocks B_1 and B_2 is $O((K/w)\log^2 z + \lceil m/z \rceil \cdot \lceil n/z \rceil \cdot z)$.

For part (iii), we compute the transposition $T_0, \ldots, T_{m-1}$ of $S_0, \ldots, S_{n-1}$, reorder the sets into $T'_0, \ldots, T'_{n-1}$ with $T'_{\pi(j)} = T_j$, and compute the transposition of $T'_0, \ldots, T'_{n-1}$. □

Theorem 2.6. *Assume $z \le N^{o(1)}$. Let $\delta > 0$ be any fixed constant and $c \in [C_1, (1/C_1)\log N/\log^2 \log N]$ for a sufficiently large constant C_1. Given a set P of N points in $d = c \log N$ dimensions, we can construct a data structure in $O(N^{1+\delta})$ preprocessing time and space, so that we can answer N offline dominance range reporting queries (with a compressed output representation) in $N^{2-1/O(c \log c)} + O(((K/w)\log^2 z + N^2/z)\log d)$ time where K is the total number of reported points over the N queries.*

Proof. We adapt the preprocessing and query algorithm in Section 2.1, with the improvement from Section 2.2. A *numbering* of a set S of n elements refers to a bijection from S to n consecutive integers. For each point set P generated by the preprocessing algorithm, we define a numbering ϕ_P of P simply by recursively "concatenating" the numberings ϕ_{P_L} and ϕ_{P_R} and appending $p(i^*)$. The output to each query for P will be a compressed representation of the subset of dominated points after applying ϕ_P.

In step 2 of the query algorithm, we can union the output for P_L and for P_R in $O(1)$ time by Lemma 2.5(i). In step 1 of the query algorithm, we need additional work since the output is with respect to a different numbering ϕ_{P_J}, for some set $J \in \mathcal{J}$. For each $J \in \mathcal{J}$, we can change the compressed representation to follow the numbering ϕ_P by invoking Lemma 2.5(iii), after collecting all query points $Q(P_J)$ that are passed to P_J (since queries are offline). To account for the cost of this invocation to Lemma 2.5(iii), we charge (a) $(1/w)\log^2 z$ units to each output feature, (b) $1/z$ units to each point pair in $P_J \times Q(P_J)$, (c) 1 unit to each point in P_J, and (d) 1 unit to each point in $Q(P_J)$, and (e) z units to the point set P_J itself.

Each output feature or each point pair is charged $O(\log d)$ times, since d decreases to $\lfloor d/2 \rfloor$ with each charge. Thus, the total cost for (a) and (b) is $O((K/w)\log^2 z \log d + (N^2/z)\log d)$. The total cost of (c) is $N^{1+o(1)}$ by the analysis of our original preprocessing algorithm; similarly, the total cost of (e) is $zN^{1+o(1)}$. The total cost of (d) is $N^{2-1/O(c \log c)}$ by the analysis of our original query algorithm.

We can make the final compressed representations to be with respect to any user-specified numbering of P, by one last invocation to Lemma 2.5(iii). The algorithm can be derandomized as in Section 2.3 (since queries are offline). □

One may wonder whether the previous range-tree-like offline algorithm by Impagliazzo et al. [18, 11] could also be adapted; the problem there is that d is only decremented rather than halved, which makes the cost of re-numbering too large.

The main application is to (min,+) matrix multiplication and all-pairs shortest paths (APSP). The corollary below follows immediately from a simple reduction [9] (see footnote 4) to d instances of d-dimensional offline dominance range reporting where the total output size K is $O(n^2)$. Here, we set $w := \delta_0 \log N$ and $z := \text{poly}(\log N)$.

Corollary 2.7. *Let $d = (1/C_1)\log^2 N/\log^2 \log N$ for a sufficiently large constant C_1. Given an $N \times d$ and a $d \times N$ real-valued matrix, we can compute their (min,+)-product (with a compressed output representation) in $O((N^2/\log N)\log^3 \log N)$ expected time.*

The corollary below follows from applying Corollary 2.7 q/d times, in conjunction with a subroutine by Chan [10, Corollary 2.5]. (The result improves [10, Corollary 2.6].)

Corollary 2.8. *Let $q = \log^3 N/\log^5 \log N$. Given an $N \times q$ and a $q \times N$ real-valued matrix, we can compute their (min,+)-product in $O(N^2)$ time.*

Applying Corollary 2.8 N/q times (and using a standard reduction from APSP to (min,+)-multiplication), we obtain:

Corollary 2.9. *Given two $N \times N$ real-valued matrices, we can compute their (min,+)-product by a combinatorial algorithm in $O((N^3/\log^3 N)\log^5 \log N)$ time. Consequently, we obtain a combinatorial algorithm for APSP for arbitrary N-vertex real-weighted graphs with the same time bound.*

Note that Williams' algorithm [27] is faster (achieving $N^3/2^{\Omega(\sqrt{\log N})}$ time), but is non-combinatorial and gives a worse time bound ($O(N^2 \log^{O(1)} N)$) for the rectangular matrix case in Corollary 2.8.

2.5 Simplified Boolean Version, with Application to BMM

In this subsection, we note that our data structure in Section 2.1 can be much simplified in the Boolean case when the input coordinates are all 0's and 1's.

The data structure is essentially an augmented, randomized variant of the standard *trie*.

Data structure. Let $\delta \in (0,1)$ and $c \in [\delta C_0, (\delta/C_0)\log N/\log^3 \log N]$ be user-specified parameters. Let b be a parameter to be chosen later.

Given a set P of $n \le N$ Boolean data points in $d \le c \log N$ dimensions, our data structure is constructed as follows:

0. If $d = 0$, then return.
1. For every possible Boolean query point with at most d/b 1's, store its answer in a table. The number of table entries is $O\left(\binom{d}{\lfloor d/b \rfloor}\right) = b^{O(d/b)}$.
2. Pick a random $i^* \in \{1, \ldots, d\}$. Recursively construct data structures for
   - the subset P_L of all points in P with i^*-th coordinate 0, and
   - the subset P_R of all points in P with i^*-th coordinate 1,

   dropping the i^*-th coordinates in both sets.

Analysis. The preprocessing time and space satisfy the recurrence

$$T_d(n) \le \max_{\alpha\in[0,1]} \left[T_{d-1}(\alpha n) + T_{d-1}((1-\alpha)n) + b^{O(d/b)}n\right],$$

which solves to $T_d(N) \le db^{O(d/b)}N \le dN^{1+O((c/b)\log b)} = dN^{1+O(\delta)}$ by setting $b := (c/\delta)\log(c/\delta)$.

Query algorithm. Given the preprocessed set P and a query point $q = (q_1, \ldots, q_d)$, our query algorithm proceeds as follows:

0. If $d = 0$, then return the answer directly.
1. If q has at most d/b 1's, then return the answer from the table.
2. Otherwise,
   - if $q_{i^*} = 1$, then recursively answer the query for P_L and for P_R (dropping the i^*-th coordinate of q);
   - if $q_{i^*} = 0$, then recursively answer the query for P_L only.

Analysis. We assume that the query point q is independent of the random choices made during the preprocessing of P. If q has more than d/b 1's, then the probability that we make a recursive call for P_R is at most $1 - 1/b$. Say that the number of points in P_L is αn. The expected number of leaves in the recursion (ignoring trivial subproblems with $n = 0$) satisfies the following recurrence:

$$Q_d(n) \ \le \ \max_{0\le\alpha\le1} \left[\left(1-\frac{1}{b}\right)Q_{d-1}(\alpha n) + Q_{d-1}((1-\alpha)n)\right], \tag{3}$$

with $Q_d(0) = 0$ and $Q_0(n) = 1$ for the base cases.

We guess that

$$Q_d(n) \le (1+\gamma)^d n^{1-\varepsilon}$$

for some choice of parameters $\gamma, \varepsilon \in (0, 1/2)$. We verify the guess by induction.

Assume that the guess is true for dimension $d-1$. Let α be the value that attains the maximum in (3). Then

$$\begin{aligned} Q_d(n) \ &\le \ \left[\left(1-\frac{1}{b}\right)\alpha^{1-\varepsilon} + (1-\alpha)^{1-\varepsilon}\right](1+\gamma)^{d-1}n^{1-\varepsilon} \\ &\le \ (1+\gamma)^d n^{1-\varepsilon}, \end{aligned}$$

provided that we can upper-bound the following expression by $1+\gamma$:

$$\left(1-\frac{1}{b}\right)\alpha^{1-\varepsilon} + (1-\alpha)^{1-\varepsilon}. \tag{4}$$

The proof is split into two cases. If $\alpha \le \gamma^2$, then (4) is at most $\alpha^{1-\varepsilon} + 1 \le 1 + \gamma$. If $\alpha > \gamma^2$, then (4) is at most

$$\begin{aligned} \left(1-\frac{1}{b}\right)\alpha e^{\varepsilon\ln(1/\alpha)} + 1 - (1-\varepsilon)\alpha \ &\le \ \left(1-\frac{1}{b}\right)\alpha(1+3\varepsilon\log(1/\gamma)) + 1 - (1-\varepsilon)\alpha \\ &\le \ 1 - \frac{\alpha}{b} + 4\alpha\varepsilon\log(1/\gamma), \end{aligned}$$

which is at most 1 by setting $\varepsilon := 1/(4b\log(1/\gamma))$.

We can set $\gamma := 1/b^3$, for example. Then $\varepsilon = O(1/(b\log b))$. We conclude that

$$Q_d(N) \;\le\; (1+\gamma)^d N^{1-\varepsilon} \;\le\; e^{\gamma d} N^{1-\varepsilon} \;\le\; N^{1-\varepsilon+O(\gamma c)} \;\le\; N^{1-1/O(b\log b)} \;\le\; N^{1-1/O((c/\delta)\log^2(c/\delta))}.$$

Now, $Q_d(N)$ excludes the cost at internal nodes of the recursion. Since the recursion has depth at most d and each internal node has $O(1)$ cost, the actual expected query time can be bounded by $O(dQ_d(N))$, which is $N^{1-1/O((c/\delta)\log^2(c/\delta))}$ for $c/\delta \ll \log N/\log^2\log N$.

This result in itself is no better than our result for the general dominance problem. However, the simplicity of the algorithm makes it easier to bit-pack the output (than the algorithm in Section 2.4):

Theorem 2.10. *Let $\delta > 0$ be any fixed constant and $c \in [C_1, (1/C_1)\log N/\log^3\log N]$ for a sufficiently large constant C_1. Given a set P of N Boolean points in $d = c\log N$ dimensions, we can construct a data structure in $O(N^{1+\delta})$ preprocessing time and space, so that we can answer N offline dominance range reporting queries (with output represented as bit vectors) in $N^{2-1/O(c\log c)} + O((N^2/w)\log w)$ time.*

Proof. We define a numbering ϕ_P of P simply by recursively concatenating the numberings ϕ_{P_L} and ϕ_{P_R}. In the table, we store each answer as a bit vector, with respect to this numbering ϕ_P. Each query can then be answered in $N^{1-1/O(c\log c)} + O(N/w)$ time, including the cost of outputting.

One issue remains: the outputs are bit vectors with respect to a particular numbering ϕ_P. To convert them into bit vectors with respect to any user-specified numbering, we first form an $N \times N$ matrix from these N (row) bit vectors (since queries are offline); we transpose the matrix, then permute the rows according to the new numbering, and transpose back. Matrix transposition can be done in $O((N^2/w)\log w)$ time, since each $w \times w$ submatrix can be transposed in $O(\log w)$ time [24]. The algorithm can be derandomized as in Section 2.3 (since queries are offline). $\square$

Since Boolean dot product is equivalent to Boolean dominance testing (see footnote 2), we immediately obtain a new result on rectangular Boolean matrix multiplication (with $w := \delta_0 \log N$):

Corollary 2.11. *Let $d = (1/C_1)\log^2 N/\log^3\log N$ for a sufficiently large constant C_1. Given an $N \times d$ and a $d \times N$ Boolean matrix, we can compute their Boolean product by a combinatorial algorithm in $O((N^2/\log N)\log\log N)$ time.*

Applying the above corollary N/d times yields a combinatorial algorithm for multiplying two $N \times N$ Boolean matrices in $O((N^3/\log^3 N)\log^4\log N)$ time. This is no better than Chan's previous BMM algorithm [11], which in turn is a logarithmic factor worse than Yu's algorithm [28], but neither previous algorithm achieves subquadratic time for the particular rectangular matrix case in Corollary 2.11.

3 Offline Dominance Range Searching

In this section, we study the offline orthogonal range searching problem in the counting version (which includes the detection version), allowing the use of fast matrix multiplication. By doubling the dimension (footnote 1), it suffices to consider the dominance case: given n data/query points in $\mathbb{R}^d$, we want to count the number of data points dominated by each query point. We describe a black-box reduction of the real case to the Boolean case.

We use a recursion similar to a degree-s range tree (which bears some resemblance to a low-dimensional algorithm from [13]).

Algorithm. Let $\delta \in (0,1)$ and s be parameters to be set later. Let $[s]$ denote $\{0, 1, \ldots, s-1\}$.

Given a set P of $n \leq N$ data/query points in $\mathbb{R}^j \times [s]^{d-j}$, with $d \leq c \log N$, our algorithm is simple and proceeds as follows:

0. If $j = 0$, then all points are in $[s]^d$ and we solve the problem directly by mapping each point $(p_1, \ldots, p_d)$ to a binary string $1^{p_1} 0^{s-p_1} \cdots 1^{p_d} 0^{s-p_d} \in \{0,1\}^{ds}$ and running a known Boolean offline dominance algorithm in ds dimensions.

1. Otherwise, for each $i \in [s]$, recursively solve the problem for the subset P_i of all points in P with ranks from $i(n/s)+1$ to $(i+1)(n/s)$ in the j-th coordinate.

2. "Round" the j-th coordinate values of all data points in P_i to $i+1$ and all query points in P_i to i, and recursively solve the problem for P after rounding (which now lies in $\mathbb{R}^{j-1} \times [s]^{d-j+1}$); add the results to the existing counts of all the query points.

Analysis. Suppose that the Boolean problem for n points in $d \leq c \log n$ dimensions can be solved in $d^C n^{2-f(c)}$ time for some absolute constant $C \geq 1$ and some function $f(c) \in [0, 1/4]$. The following recurrence bounds the total cost of the leaves of the recursion in our algorithm (assuming that n is a power of s, for simplicity):

$$T_{d,j}(n) = s\, T_{d,j}(n/s) + T_{d,j-1}(n).$$

For the base cases, $T_{d,j}(1) = 1$; and if $n > \sqrt{N}$, then $T_{d,0}(n) \leq (ds)^C n^{2-f(2cs)}$ (since the Boolean subproblems have dimension $ds \leq cs \log N \leq 2cs \log n$). On the other hand, if $n \leq \sqrt{N}$, we can use brute force to get $T_{d,0}(n) \leq dn^2 \leq dn^{3/2} N^{1/4}$. In any case, $T_{d,0}(n) \leq (ds)^C n^{3/2} N^{1/2-f(2cs)} = An^{3/2}$ where we let $A := (ds)^C N^{1/2-f(2cs)}$.

One way[5] to solve this recurrence is again by "guessing". We guess that

$$T_{d,j}(n) \leq (1+\gamma)^j A n^{3/2}$$

for some choice of parameter $\gamma \in (0,1)$ to be determined later. We verify the guess by induction.

The base cases are trivial. Assume that the guess is true for lexicographically smaller (j, n). Then

$$\begin{aligned} T_{d,j}(n) &\leq (1+\gamma)^j A s (n/s)^{3/2} + (1+\gamma)^{j-1} A n^{3/2} \\ &= \left[\frac{1}{\sqrt{s}} + \frac{1}{1+\gamma}\right] (1+\gamma)^j A n^{3/2} \\ &\leq (1+\gamma)^j A n^{3/2}, \end{aligned}$$

provided that

$$\frac{1}{\sqrt{s}} + \frac{1}{1+\gamma} \leq 1,$$

which is true by setting $\gamma := 2/\sqrt{s}$.

[5] Since this particular recurrence is simple enough, an alternative, more direct way is to expand $T_{d,d}(N)$ into a sum $\sum_{i \geq 0} \binom{d+i}{i} s^i T_{d,0}(N/s^i) \leq \sum_{i \geq 0} O(\frac{d+i}{i\sqrt{s}})^i \cdot AN^{3/2}$, and observe that the maximum term occurs when i is near $d/\sqrt{s}$...

We can set $s := c^4$, for example. Then $\gamma = O(1/c^2)$. We conclude that

$$\begin{aligned} T_{d,d}(N) &\leq (1+\gamma)^d A N^{3/2} \leq e^{\gamma d}(ds)^{O(1)} N^{2-f(2cs)} \\ &\leq (ds)^{O(1)} N^{2-f(2cs)+O(\gamma c)} = d^{O(1)} N^{2-f(2c^5)+O(1/c)}. \end{aligned}$$

Now, $T_{d,d}(N)$ excludes the cost at internal nodes of the recursion. Since the recursion has depth at most $\log_s N + d$, the actual running time can be bounded by $T_{d,d}(n)(d\log N)^{O(1)}$.

Abboud, Williams, and Yu's algorithm [1] for the Boolean case, as derandomized by Chan and Williams [14], achieves $f(c) = 1/O(\log c)$, yielding an overall time bound of $N^{2-1/O(\log c)}(d\log N)^{O(1)}$, which is $N^{2-1/O(\log c)}$ for $\log c \ll \sqrt{\log N}$.

Theorem 3.1. *Let $c \in [1, 2^{(1/C_1)\sqrt{\log N}}]$ for a sufficiently large constant C_1. Given N points in $d = c\log N$ dimensions, we can answer N offline dominance range counting queries in $N^{2-1/O(\log c)}$ time.*

We remark that if the Boolean problem could be solved in truly subquadratic time $d^{O(1)}N^{2-\varepsilon}$, then the above analysis (with $s := (c\log N)^2$, say) would imply that the general problem could be solved in truly subquadratic time with the *same* ε, up to $(d\log N)^{O(1)}$ factors.

4 Approximate ℓ_∞ Nearest Neighbor Searching

In this section, we study (online, combinatorial) data structures for t-approximate ℓ_∞ nearest neighbor search. By known reductions [17, 19], it suffices to solve the fixed-radius approximate decision problem, say, for radius $r = 1/2$: given a query point q, we want to find a data point of distance at most distance $t/2$ from q, under the promise that the nearest neighbor distance is at most $1/2$.

Our solution closely follows Indyk's divide-and-conquer method [19], with a simple modification that incorporates a range-tree-like recursion.

4.1 Main Data Structure

Data structure. Let $\delta \in (0,1)$, $\rho > 1$, and $c \geq 4$ be user-specified parameters. Let s and k be parameters to be chosen later.

Given a set P of $n \leq N$ data points in $d \leq c\log N$ dimensions, our data structure is constructed as follows:

0. If $n \leq s$ or $d = 0$, then just store the points in P.

   Otherwise, compute and store the median first coordinate μ in P. Let $P_{>i}$ (resp. $P_{<i}$) denote the subset of all points in P with first coordinate greater than (resp. less than) $\mu + i$. Let $\alpha_i := |P_{>i}|/n$ and $\beta_i := |P_{<-i}|/n$. Note that the α_i's and β_i's are decreasing sequences with $\alpha_0 = \beta_0 = 1/2$.

1. If $\alpha_k > 1/s$ and $\alpha_{i+1} > \alpha_i^\rho$ for some $i \in \{0, 1, \ldots, k-1\}$, then set *type* $= (1, i)$ and recursively construct a data structure for $P_{>i}$ and for $P_{<i+1}$.

2. Else if $\beta_k > 1/s$ and $\beta_{i+1} > \beta_i^\rho$ for some $i \in \{0, 1, \ldots, k-1\}$, then set *type* $= (2, i)$ and recursively construct a data structure for $P_{<-i}$ and for $P_{>-(i+1)}$.

3. Else if $\alpha_k, \beta_k \le 1/s$, then set *type* $= 3$ and recursively construct a data structure for
    - the set $P_{>k} \cup P_{<-k}$ and
    - the $(d-1)$-dimensional projection of $P - (P_{>k+1} \cup P_{<-(k+1)})$ that drops the first coordinate (this recursion in $d-1$ dimensions is where our algorithm differs from Indyk's).

We set $k := \lceil \log_\rho \log s \rceil$. Then one of the tests in steps 1–3 must be true. To see this, suppose that $\alpha_k > 1/s$ (the scenario $\beta_k > 1/s$ is symmetric), and suppose that i does not exist in step 1. Then $\alpha_k \le (1/2)^{\rho^k} \le 1/s$, a contradiction.

Analysis. The space usage is proportional to the number of points stored at the leaves in the recursion, which satisfies the following recurrence (by using the top expression with $(\alpha, \alpha') = (\alpha_i, \alpha_{i+1})$ for step 1 or $(\alpha, \alpha') = (\beta_i, \beta_{i+1})$ for step 2, or the bottom expression for step 3):

$$S_d(n) \le \max \begin{cases} \displaystyle\max_{\alpha,\alpha':\ \alpha' > \alpha^\rho,\ 1/s < \alpha' \le \alpha \le 1/2} \left[S_d(\alpha n) + S_d((1-\alpha')n) \right] \\ S_d(2n/s) + S_{d-1}(n), \end{cases} \tag{5}$$

with $S_d(n) = n$ for the base case $n \le s$ or $d = 0$.

We guess that

$$S_d(n) \le (1+\gamma)^d n^\rho$$

for some choice of parameter $\gamma \in (0,1)$. We verify the guess by induction.

The base case is trivial. Assume that the guess is true for lexicographically smaller (d, n).

- Case I: the maximum in (5) is attained by the top expression and by α, α'. Then

$$\begin{aligned} S_d(n) &\le (1+\gamma)^d \left[(\alpha n)^\rho + ((1-\alpha')n)^\rho \right] \\ &\le \left[\alpha^\rho + 1 - \alpha' \right] (1+\gamma)^d n^\rho \\ &\le (1+\gamma)^d n^\rho. \end{aligned}$$

- Case II: the maximum in (5) is attained by the bottom expression. Then

$$\begin{aligned} S_d(n) &\le (1+\gamma)^d (2n/s)^\rho + (1+\gamma)^{d-1} n^\rho \\ &\le \left[\left(\frac{2}{s}\right)^\rho + \frac{1}{1+\gamma} \right] (1+\gamma)^d n^\rho \\ &\le (1+\gamma)^d n^\rho \end{aligned}$$

  by setting $s := 2(2/\gamma)^{1/\rho}$.

Set $\gamma := \delta/c$. Then $s = O((c/\delta)^{1/\rho})$ and $k = \log_\rho \log(c/\delta) + O(1)$. We conclude that

$$S_d(N) \ \le \ e^{\gamma d} N^\rho \ \le \ N^{\rho + O(\gamma c)} \ = \ N^{\rho + O(\delta)}.$$

For the preprocessing time, observe that the depth of the recursion is $h := O(\log_{s/(s-1)} N + d)$ (since at each recursive step, the size of the subsets drops by a factor of $1 - 1/s$ or the dimension decreases by 1). Now, $h = O(s \log N + d) \le O((c/\delta) \log N + d) = O((c/\delta) \log N)$. Hence, the preprocessing time can be bounded by $O(S_d(N)h) = O((c/\delta) N^{\rho+\delta} \log N)$.

Query algorithm. Given the preprocessed set P and a query point $q = (q_1, \ldots, q_d)$, our query algorithm proceeds as follows:

0. If $n \le s$ or $d = 0$, then answer the query directly by brute-force search.
1. If *type* $= (1, i)$: if $q_1 > i + 1/2$, then recursively answer the query in $P_{>i}$, else recursively answer the query in $P_{<i+1}$.
2. If *type* $= (2, i)$: proceed symmetrically.
3. If *type* $= 3$:
   - if $q_1 > k + 1/2$ or $q_1 < -(k + 1/2)$, then recursively answer the query in $P_{>k} \cup P_{<-k}$;
   - else recursively answer the query in $P - (P_{>k+1} \cup P_{<-(k+1)})$, after dropping the first coordinate of q.

Note that in the last subcase of step 3, any returned point has distance at most $2k + 3/2$ from q in terms of the first coordinate. By induction, the approximation factor t is at most $4k + 3 = O(\log_\rho \log(c/\delta))$.

Analysis. The query time is clearly bounded by the depth h, which is $O((c/\delta) \log N)$.

Theorem 4.1. *Let $\delta > 0$ be any fixed constant. Let $\rho > 1$ and $c \ge \Omega(1)$. Given N points in $d = c \log N$ dimensions, we can construct a data structure in $O(dN^{\rho+\delta})$ time and $O(dN + N^{\rho+\delta})$ space, so that we can handle the fixed-radius decision version of approximate ℓ_∞ nearest neighbor queries in $O(d)$ time with approximation factor $O(\log_\rho \log c)$.*

4.2 Linear-Space Version

In this subsection, we describe a linear-space variant of the data structure in Section 4.1. (To our knowledge, a linear-space variant of Indyk's data structure [19], which our solution is based on, has not been reported before.) The approximation factor is unfortunately poorer, but the fact that the data structure is just a plain constant-degree tree with N leaves may be attractive in certain practical settings.

The high-level idea of the variant is simple: in step 1 or 2, instead of recursively generating two subsets that may overlap, we partition into three *disjoint* subsets; this guarantees linear space but increases the cost of querying.

Data structure. Let $\rho, \delta \in (0, 1)$ be user-specified parameters. Let s and k be parameters to be chosen later. Given a set P of $n \le N$ data points in $d \le c \log N$ dimensions, our data structure is constructed as follows:

0. If $n \le s$ or $d = 0$, then just store the points in P.

   Otherwise, compute the median first coordinate μ in P. Let $P_{>i}$ (resp. $P_{<i}$) denote the subset of all points in P with first coordinate greater than (resp. less than) $\mu + i$. Let $\alpha_i := |P_{>i}|/n$ and $\beta_i := |P_{<-i}|/n$. Define the function $f(\alpha) := \alpha - (\rho\alpha)^{1/\rho}$.

1. If $\alpha_k > 1/s$ and $\alpha_{i+1} > f(\alpha_i)$ for some $i \in \{0, 1, \ldots, k-1\}$, then set *type* $= (1, i)$ and recursively construct a data structure for $P_{>i+1}$, for $P - (P_{>i+1} \cup P_{<i})$, and for $P_{<i}$.
2. If $\beta_k > 1/s$ and $\beta_{i+1} > f(\beta_i)$ for some $i \in \{0, 1, \ldots, k-1\}$, then set *type* $= (2, i)$ and recursively construct a data structure for $P_{<-(i+1)}$, for $P - (P_{<-(i+1)} \cup P_{>-i})$, and for $P_{>-i}$.
3. If $\alpha_k, \beta_k \leq 1/s$, then set *type* $= 3$ and recursively build a data structure for
   - the set $P_{>k} \cup P_{<-k}$ and
   - the $(d-1)$-dimensional projection of $P - (P_{>k} \cup P_{<-k})$ that drops the first coordinate.

We set $k := \left\lceil \frac{2\rho}{1-\rho} s^{(1-\rho)/\rho} \right\rceil$. Then one of the tests in steps 1–3 must be true by the following lemma:

Lemma 4.2. *For any sequence $\alpha_0, \alpha_1, \ldots \in [0,1]$ with $\alpha_{i+1} \leq \alpha_i - (\rho\alpha_i)^{1/\rho}$ and $\alpha_k > 1/s$, we have $k < \frac{2\rho}{1-\rho} s^{(1-\rho)/\rho}$.*

Proof. Let $t := (1-\rho)/\rho$. Then

$$\alpha_{i+1}^{-t} \;\geq\; \left[\alpha_i(1 - \rho^{1/\rho}\alpha_i^{1/\rho - 1})\right]^{-t} \;\geq\; \alpha_i^{-t}\left(1 + t\rho^{1/\rho}\alpha_i^{1/\rho-1}\right) \;\geq\; \alpha_i^{-t} + t\rho^{1/\rho}.$$

Iterating k times yields $\alpha_k^{-t} \geq (t\rho^{1/\rho})k \geq tk/2$. Thus, $k \leq (2/t)\alpha_k^{-t} < (2/t)s^t$. □

Analysis. Space usage is clearly linear. Since the depth of the recursion is $h = O(\log_{s/(s-1)} N + d)$, the preprocessing time can be bounded by $O(Nh) = O(sN \log N + dN)$.

Query algorithm. Given the preprocessed set P and a query point $q = (q_1, \ldots, q_d)$, our query algorithm proceeds as follows:

0. If $n \leq s$ or $d = 0$, then answer the query directly by brute-force search.
1. If *type* $= (1, i)$: if $q_1 > i + 1/2$, then recursively answer the query in $P_{>i+1}$ and in $P - (P_{>i+1} \cup P_{<i})$; else recursively answer the query in $P_{<i}$ and in $P - (P_{>i+1} \cup P_{<i})$.
2. If *type* $= (2, i)$: proceed symmetrically.
3. If *type* $= 3$:
   - if $q_1 > k + 1/2$ or $q_1 < -(k + 1/2)$, then recursively answer the query in $P_{>k} \cup P_{<-k}$;
   - else recursively answer the query in $P_{>k} \cup P_{<-k}$ and in $P - (P_{>k} \cup P_{<-k})$, after dropping the first coordinate of q for the latter.

Note that in the last recursive call in step 3, any returned point has the distance at most $2k + 1/2$ from q with respect to the first coordinate. By induction, the approximation factor t is at most $4k + 1$.

Analysis. The query time is bounded by $O(s)$ times the number of leaves in the recursion, which satisfies the following recurrence (by using the top expression with $(\alpha, \alpha') = (\alpha_i, \alpha_{i+1})$ for step 1 or $(\alpha, \alpha') = (\beta_i, \beta_{i+1})$ for step 2, or the bottom expression for step 3):

$$Q_d(n) \leq \max \begin{cases} \displaystyle\max_{\alpha,\alpha' \colon\ \alpha' > f(\alpha),\ 1/s < \alpha' \leq \alpha \leq 1/2} \left[Q_d(\max\{\alpha' n, (1-\alpha)n\}) + Q_d((\alpha - \alpha')n)\right] \\ Q_d(2n/s) + Q_{d-1}(n), \end{cases} \tag{6}$$

with $Q_d(n) = 1$ for the base case $n \leq s$ or $d = 0$.

We guess that

$$Q_d(n) \leq (1+\gamma)^d n^\rho$$

for some choice of parameter $\gamma \in (0, 1)$. We verify the guess by induction.

The base case is trivial. Assume that the guess is true for lexicographically smaller (d, n).

- Case I: the maximum in (6) is attained by the top expression and by α, α'. Then

  $$\begin{aligned} Q_d(n) &\leq (1+\gamma)^d \left[((1-\alpha)n)^\rho + ((\alpha - \alpha')n)^\rho\right] \\ &\leq \left[1 - \rho\alpha + (\alpha - \alpha')^\rho\right](1+\gamma)^d n^\rho \\ &\leq (1+\gamma)^d n^\rho, \end{aligned}$$

  since $\alpha' > \alpha - (\rho\alpha)^{1/\rho}$.

- Case II: the maximum in (6) is attained by the bottom expression. Then

  $$\begin{aligned} Q_d(n) &\leq (1+\gamma)^d (2n/s)^\rho + (1+\gamma)^{d-1} n^\rho \\ &\leq \left[\left(\frac{2}{s}\right)^\rho + \frac{1}{1+\gamma}\right](1+\gamma)^d n^\rho \\ &\leq (1+\gamma)^d n^\rho \end{aligned}$$

  by setting $s := 2(2/\gamma)^{1/\rho}$.

Set $\gamma := \delta/c$. Then $s = O(c/\delta)^{1/\rho}$ and $k = O(\frac{\rho}{1-\rho}) \cdot O(c/\delta)^{(1-\rho)/\rho^2}$. We conclude that

$$Q_d(N) \ \leq\ e^{\gamma d} N^\rho \ \leq\ N^{\rho + O(\gamma c)} = N^{\rho + O(\delta)}.$$

Theorem 4.3. *Let $\delta > 0$ be any fixed constant. Let $\rho \in (\delta, 1-\delta)$ and $c \geq \Omega(1)$. Given N points in $d = c \log N$ dimensions, we can construct a data structure in $O(dN)$ time and space, so that we can handle the fixed-radius decision version of approximate ℓ_∞ nearest neighbor queries in $O(N^{\rho+\delta})$ time with approximation factor $O(c^{(1-\rho)/\rho^2})$.*

References

[1] A. Abboud, R. Williams, and H. Yu. More applications of the polynomial method to algorithm design. In *Proc. 26th ACM–SIAM Sympos. Discrete Algorithms (SODA)*, pages 218–230, 2015.

[2] P. Afshani, T. M. Chan, and K. Tsakalidis. Deterministic rectangle enclosure and offline dominance reporting on the RAM. In *Proc. 41st Int. Colloq. Automata, Languages, and Programming (ICALP), Part I*, pages 77–88, 2014.

[3] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Inform. Comput.*, 136(1):25–51, 1997.

[4] J. Alman, T. M. Chan, and R. Williams. Polynomial representation of threshold functions with applications. In *Proc. 57th IEEE Sympos. Found. Comput. Sci. (FOCS)*, pages 467–476, 2016.

[5] J. Alman and R. Williams. Probabilistic polynomials and Hamming nearest neighbors. In *Proc. 56th IEEE Sympos. Found. Comput. Sci. (FOCS)*, pages 136–150, 2015.

[6] A. Andoni, D. Croitoru, and M. M. Pătraşcu. Hardness of nearest neighbor under L_∞. In *Proc. 49th IEEE Sympos. Found. Comput. Sci. (FOCS)*, pages 424–433, 2008.

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

[8] T. M. Chan. Geometric applications of a randomized optimization technique. *Discrete Comput. Geom.*, 22(4):547–567, 1999.

[9] T. M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica*, 50:236–243, 2008.

[10] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM J. Comput.*, 39:2075–2089, 2010.

[11] T. M. Chan. Speeding up the Four Russians algorithm by about one more logarithmic factor. In *Proc. 26th ACM–SIAM Sympos. Discrete Algorithms (SODA)*, pages 212–217, 2015.

[12] T. M. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th ACM Sympos. Comput. Geom. (SoCG)*, pages 1–10, 2011.

[13] T. M. Chan and M. Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proc. 21st ACM–SIAM Sympos. Discrete Algorithms (SODA)*, pages 161–173, 2010.

[14] T. M. Chan and R. Williams. Deterministic APSP, orthogonal vectors, and more: Quickly derandomizing Razborov–Smolensky. In *Proc. 27th ACM–SIAM Sympos. Discrete Algorithms (SODA)*, pages 1246–1255, 2016.

[15] D. M. Gordon, O. Patashnik, G. Kuperberg, and J. Spencer. Asymptotically optimal covering designs. *J. Combinatorial Theory, Series A*, 75(2):270–280, 1996.

[16] Y. Han and T. Takaoka. An $O(n^3 \log\log n/\log^2 n)$ time algorithm for all pairs shortest paths. In *Proc. 13th Scand. Sympos. and Workshops on Algorithm Theory (SWAT)*, pages 131–141, 2012.

[17] S. Har-Peled, P. Indyk, and R. Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory Comput.*, 8(1):321–350, 2012.

[18] R. Impagliazzo, S. Lovett, R. Paturi, and S. Schneider. 0-1 integer linear programming with a linear number of constraints. arXiv:1401.5512, 2014.

[19] P. Indyk. On approximate nearest neighbors under l_∞ norm. *J. Comput. Sys. Sci.*, 63(4):627–638, 2001.

[20] K. G. Larsen and R. Williams. Faster online matrix-vector multiplication. In *Proc. 28th ACM–SIAM Sympos. Discrete Algorithms (SODA)*, pages 2182–2189, 2017.

[21] F. Le Gall. Faster algorithms for rectangular matrix multiplication. In *Proc. 53rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 514–523, 2012.

[22] J. Matoušek. Computing dominances in E^n. *Inform. Process. Lett.*, 38(5):277–278, 1991.

[23] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer–Verlag, 1985.

[24] M. Thorup. Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise Boolean operations. *J. Algorithms*, 42:205–230, 2002.

[25] R. Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005.

[26] R. Williams. Matrix-vector multiplication in sub-quadratic time (some preprocessing required). In *Proc. 18th ACM–SIAM Sympos. Discrete Algorithms (SODA)*, pages 995–1001, 2007.

[27] R. Williams. Faster all-pairs shortest paths via circuit complexity. In *Proc. 46th ACM Sympos. Theory Comput. (STOC)*, pages 664–673, 2014.

[28] H. Yu. An improved combinatorial algorithm for Boolean matrix multiplication. In *Proc. 42nd Int. Colloq. Automata, Languages, and Programming (ICALP), Part I*, pages 1094–1105, 2015.