Convexity Helps Iterated Search in 3D

Peyman Afshani ✉ 🆔
Department of Computer Science, Aarhus University, Denmark

Yakov Nekrich ✉ 🆔
Department of Computer Science, Michigan Technological University, Houghton, MI, USA

Frank Staals ✉ 🆔
Department of Information and Computing Sciences, Utrecht University, The Netherlands

── Abstract ──

Inspired by the classical fractional cascading technique [13, 14], we introduce new techniques to speed up the following type of iterated search in 3D: The input is a graph $\mathbf{G}$ with bounded degree together with a set H_v of 3D hyperplanes associated with every vertex of v of $\mathbf{G}$. The goal is to store the input such that given a query point $q \in \mathbb{R}^3$ and a connected subgraph $\mathbf{H} \subset \mathbf{G}$, we can decide if q is below or above the lower envelope of H_v for every $v \in \mathbf{H}$. We show that using linear space, it is possible to answer queries in roughly $O(\log n + |\mathbf{H}|\sqrt{\log n})$ time which improves trivial bound of $O(|\mathbf{H}| \log n)$ obtained by using planar point location data structures. Our data structure can in fact answer more general queries (it combines with shallow cuttings) and it even works when $\mathbf{H}$ is given one vertex at a time. We show that this has a number of new applications and in particular, we give improved solutions to a set of natural data structure problems that up to our knowledge had not seen any improvements.

We believe this is a very surprising result because obtaining similar results for the planar point location problem was known to be impossible [15].

1 Introduction

The idea of "iterated searching" was introduced with the classical fractional cascading technique [13, 14] and it can be described as follows: consider an abstract data structure problem where given an input I, we need to preprocess it to answer a given q more efficiently. Now imagine that instead of having one input set, I, we have multiple independent input sets $I_1, I_2, \cdots$ and at the query time, we would like to answer the same query q on *some* of the data sets. This is typically modeled as follows: the input contains a catalog graph $\mathbf{G}$ and each vertex $v \in \mathbf{G}$ is associated with a different input set I_v. The goal is to preprocess the input, that is $\mathbf{G}$ and the sets I_v, so that given a query q and a connected subgraph $\mathbf{H} \subset \mathbf{G}$ we can find the answer to querying q on every data set I_v for each $v \in \mathbf{H}$. The trivial solution is to have a separate data structure for every I_v and to query them individually. Thus, the main research interest is to improve upon this trivial solution.
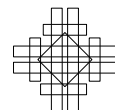
1.1 Fractional Cascading

Chazelle and Guibas improved upon the aforementioned "trivial solution" for the predecessor search problem. To be more precise, for each $v \in \mathbf{G}$, I_v is an ordered list of values and the query q is also a value and the goal is to find the predecessor of q in I_v for every $v \in \mathbf{H}$.

Chazelle and Guisbas showed how to build a data structure that uses linear space to answer such queries in $O(\log n + |\mathbf{H}|)$ time [13, 14], essentially, reducing the cost of a predecessor search to $O(1)$, after an initial investment of $O(\log n)$ time. This is clearly optimal and since its introduction has found plenty of applications.

The origin of this technique can be traced back to 1982 and to a paper by Vaishnai and Wood [22] and also the notion of "downpointers" from a 1985 paper by Willard [23] but the seminal work in this area was the aforementioned work by Chazelle and Guibas [13, 14].

1.2   2D Fractional Cascading

Due to its numerous applications, Chazelle and Liu [15] studied "2D fractional cascading" which is the iterated 2D point location problem. In particular, each vertex $v \in \mathbf{G}$ is associated with a planar subdivision, the query q is a point in $\mathbb{R}^2$, and the goal is to output for every $v \in \mathbf{H}$ the cell of the subdivion of v that contains q. Chazelle and Liu showed that in this case all hope is lost because there is an almost quadratic space lower bound for data structures with $O(\log n + |\mathbf{H}|)$ query time. Actually, their techniques prove an $\Omega(n^{2-\varepsilon})$ space lower bound for data structures with $O(\log^{O(1)} n) + o(|\mathbf{H}| \log n)$ query time. As $O(|\mathbf{H}| \log n)$ is the trivial upper bound, the lower bound "dashes all such hopes" (as Chazelle and Liu put it) for obtaining a general technique in dimensions two and beyond. However, the lower bound only applies to non-orthogonal subdivisions and only for data structures for sub-quadratic space consumption. Relatively recently, the barrier has been overcome via exploring those two directions. Afshani and Cheng [3] studied various cases involving orthogonal subdivisions and obtained almost tight results, e.g., the problem can be solved using linear space and with roughly $O(\log n + |\mathbf{H}|\sqrt{\log n})$ query time for axis-aligned subdivisions. Chan and Zheng [11] studied the non-orthogonal version but they allowed quadratic space usage and they showed that queries can be answered in $O(\log n + |\mathbf{H}|)$ time. Finally, we need to mention that a more general form of iterated search which allows $\mathbf{H}$ to be any subgraph of $\mathbf{G}$ has also been studied for 2D queries but the bounds are weaker [7].

1.3   A Brief Statement of Our Results

We introduce a general technique to solve the iterated search problem for a certain 3D query problem. Then, we show that our technique has a number of non-trivial applications.

1.3.1   A General Framework for Iterated Searching for Lower Envelopes

The exact statement of our technique requires familiarity with concepts such as shallow cuttings and therefore we postpone it to after the "Preliminaries" section. Here, we only state it informally: we study the *lower envelope iterated search (LEIS) problem*, where we are given a catalog graph $\mathbf{G}$ and each vertex $v \in \mathbf{G}$ is associated with a catalog which is a set H_v of hyperplanes. The goal is to preprocess the input into a data structure such that given a query point q and a connected subgraph $\mathbf{H} \subset \mathbf{G}$, the data structure should report whether q is below or above the lower envelope of H_v, for all $v \in \mathbf{H}$. In the shallow cutting variant of LEIS, the input has a parameter k and the data structure stores a number of lists of size $O(k)$ explicitly. Then at the query time the data structure has two options: (i) report that q is above the k-level of H_v or (ii) return a constant number, c, of pointers to c stored lists $L_1, \cdots, L_c$ such that the union of $L_1, \cdots, L_c$ contains all the hyperplanes that pass below q. In the *on-demand* version of the problem, $\mathbf{H}$ is given one vertex v at a time such that all the given vertices form a connected subgraph of $\mathbf{G}$ but the query must answer q on v before the next vertex is given. See Figure 1 for an illustration of our problem.
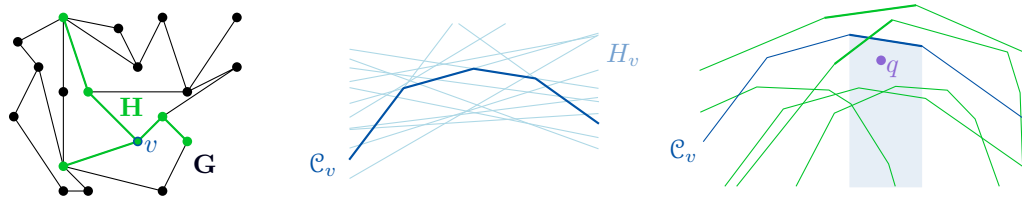
Figure 1 A schematic drawing of the LEIS Problem (with hyperplanes in $\mathbb{R}^2$ rather than $\mathbb{R}^3$). Each vertex v of **G** is associated with a set of hyperplanes H_v. For each vertex v in the query subgraph $\mathbf{H} \subset \mathbf{G}$ we wish to find the triangle in the k-shallow cutting $\mathcal{C}_v$ above the query point q.

Our main result is that all the above variants can be solved with a linear-space data structure that has $O(\log n + |\mathbf{H}|\sqrt{\log n})$ query time. This improves upon the trivial solution which results in $O(|\mathbf{H}| \log n)$ query time.

We believe this is a surprising result since a LEIS query (which is very commonly used in combination with shallow cuttings, see Lemma 3) is typically answered via a point location query and as Chazelle and Liu's lower bound [15] shows, it is hopeless to have an efficient data structure for iterated point location queries.

1.3.2 Some Applications of Our Technique

Chazelle and Liu [15] cite numerous potential applications of iterated 2D point location, if such a thing were to exist, before presenting their lower bound. The lower bound applies to some of those applications but not to all of them. In particular, it turns out that for some geometric problems, we are able to give improvements. One main problem is answering *halfspace max queries* which currently has only a basic "folklore" solution in 3D, despite being used in "blackbox" reductions in at least two general frameworks (more on this below).

▶ **Problem 1** (3D halfspace max queries). *Let P be an input set of n points in $\mathbb{R}^3$ each associated with a real-valued weight. Store them in a data structure such that given a query halfspace q one can find the point with the largest weight inside q.*

Currently, we are only aware of a "folklore" solution that uses $O(n \log n)$ space and has the query time of $O(\log^2 n)$ [6]. Using our framework, we get the following improvements.

- 3D halfspace max queries can be answered in $O(\log^{3/2} n)$ time with $O(n \log n)$ space.
- The current best solution for *3D halfspace range sampling* uses a "blackbox" data structure for weighted range sampling queries [6]. As a result, we improve the query time of the data structure in [6] from $O(\log^2 + k)$ (for sampling k points from a given query halfspace) to $O(\log^{3/2} n + k)$ by simply plugging in our solution for 3D halfspace max queries.
- Range sampling queries can be used as a blackbox to obtain solutions for other approximate aggregate queries [4] and thus our improve carries over to such queries as well.
- We get improved results for *colored halfspace reporting*, in which we store a set of n colored points so that given a query halfspace we can efficiently report all k distinct colors of the points that appear in the halfspace. We can achieve $O(\log n \log \log n + m \log \log n)$ query time using linear space, where m is the number of colors, or $O(\log n + k \log(m/k)\sqrt{\log n})$ time using $O(n \log n)$ space. This improves the query time of current solutions by a $O(\log n)$, respectively $O(\sqrt{\log n})$ factor.
- Some weighted range reporting problems can also be solved more efficiently: (*weighted halfspace range reporting*) store a given set of weighted points such that given a query halfspace q and an interval $[w_1, w_2]$, report all the points inside q whose weight is between

w_1 and w_2. (*weighted halfspace stabbing*) This is similar to the previous problem with the role of points and halfspaces swapped.

- The general framework of [19] combines weighted reporting and max reporting to give a general solution for "top-k reporting" (report the k heaviest points). Using our results, we can obtain a data structure for 3D halfspaces with $O(\log^{3/2} n + k)$ query time. This was the result "missing" in [19] where they only list new results for $d = 2$ and $d \geq 4$.
- Range reporting can also be used to obtain "approximate counting" results and a number of general reductions are available [18, 8, 17, 1] and so we can obtain improved results for "approximate counting" version of the above weighted reporting problems.
- The *offline* version of halfspace range reporting can also be improved: given a set P of n points, where a point p_i is given with an insertion time s_i and a deletion time d_i, store them in a data structure such that given a query halfspace h and a timestamp t, we can report all the points in h that exist in time t.
- Via standard lifting maps, problems involving circles in 2D can be reduced to those involving halfspaces in 3D.

2   Preliminaries

We adopt the following notations and definitions. In $\mathbb{R}^3$, a *triangle* is the convex hull of three points whereas a *simplex* is the convex hull of four points. The Z-coordinate is assumed to be the vertical direction and a *vertical prism* $\tau^{\downarrow}$ is defined by a triangle τ and it consists of all the points that are on τ or directly below τ (wrt to the Z-coordinate). For a point $q \in \mathbb{R}^3$, $q^{\downarrow}$ represents a downward ray starting from q. Given a set A of geometric objects and another geometric object r, the *conflict list* of r wrt A, denoted by $\Diamond(A, r)$, is the subset of A that intersects R. For example, if A is a set of polytopes and r is a point, then $\Diamond(A, r)$ is the set of of polytopes that contain r or if A is a set of hyperplanes and r is a point, then $\Diamond(A, r^{\downarrow})$ is the subset of hyperplanes that pass through or below r; in this latter case, $|\Diamond(A, r^{\downarrow})|$ is denoted as the *level* of r (wrt A). For a given parameter k, $1 \leq k \leq n$, the $(\leq k)$-level of A is the set of all the points in $\mathbb{R}^3$ with level at most k and the k-level of A is defined as the boundary of the $(\leq k)$-level; in particular, the 0-level of A is the lower envelope of A.

▶ **Observation 2** ([9]). *Let τ be a triangle and let $\Delta = \tau^{\downarrow}$ be the vertical prism defined by it. Let v_1, v_2, and v_3 be the vertices of τ. Any hyperplanes that intersects Δ, passes below at least one of the vertices of τ. In other words, for any set H of hyperplanes we have*

$$\Diamond(H, \Delta) \subset \Diamond(H, v_1^{\downarrow}) \cup \Diamond(H, v_2^{\downarrow}) \cup \Diamond(H, v_3^{\downarrow}).$$

We will extensively use shallow cuttings in our techniques. By now, this is a standard tool that is used very often in various 3D range searching problems.

▶ **Lemma 3** ([10, 20, 21]). *There exists a fixed constant $\alpha > 1$ such that for any given set H of n hyperplanes in 3D and a given parameter k, $1 \leq k \leq n$, the following holds: There exists a set of $O(n/k)$ hyperplanes such that their lower envelope lies above the k-level of H but below the (αk)-level of H. The lower envelope can be triangulated and for every resulting triangle τ, one can also build the conflict list $\Diamond(H, \tau^{\downarrow})$ in $O(n \log n)$ time in total (over all triangles τ). The shallow cutting $\mathcal{C}$ is taken to be the set of all the triangles and each triangle stores a pointer to its explicitly stored conflict list.*

Finally, one can store the projection of the shallow cutting in a 2D point location data structure that uses $O(n/k)$ space such that given a query point $q \in \mathbb{R}^3$, in $O(\log(n/k))$ time one can decide whether q lies below any triangle in $\mathcal{C}$ and if it does, then one can find the triangle directly above q, denoted by $\mathcal{C}(q)$. The subset of hyperplanes passing below q is a subset of the conflict list of $\mathcal{C}(q)$, i.e., $\Diamond(H, q^{\downarrow}) \subset \Diamond(H, \mathcal{C}(q)^{\downarrow})$.

Remark. The above is a "modern" statement of shallow cuttings in 3D. These were originally discovered by Matoušek [21]. However, a simple observation by Chan [9] (Observation 2) allows one to simplify the structure of shallow cuttings into the lower envelope of some hyperplanes. We will be strongly using convexity in our techniques so this treatment is extremely crucial for our purposes.

We now define the main iterated search problems that we will be studying.

▶ **Problem 4** (LEIS3D)**.** *Let* **G** *be an input graph of constant degree, called the catalog graph, where each vertex $v \in G$ is associated with an input set H_v of hyperplanes in $\mathbb{R}^3$. Let n be the total input size and let k, $1 \le k \le n$, be a parameter given with the input. Store the input in a structure to answer queries that are given by a point $q \in \mathbb{R}^3$ and a connected subgraph $\mathbf{H} \subset \mathbf{G}$. For each vertex $v \in \mathbf{G}$, the data structure should store a k-shallow cutting of H_v. Then, for each $v \in \mathbf{H}$ at the query time, it must either (i) return a triangle τ above q together with a list of c pointers, for a constant c, to c conflict lists $L_1, \cdots, L_c$ in the k-shallow cutting of H_v, such that $\Diamond(H_v, \tau) \subset \cup_{i=1}^c L_i$ or (ii) indicate that q lies above the k-level of H_v.*

In the anchored version of the problem, for every $v \in \mathbf{G}$ the data structure must explicitly store a set T_v of triangles such that the triangle τ_v returned at query time must be an element of T_v. In the on-demand version of the problem, q is given first but $\mathbf{H}$ is initially hidden and is revealed one vertex at a time at query time. The data structure must answer the query on each revealed vertex before the next vertex can be revealed. For simplicity, we assume that $\mathbf{H}$ is a walk, i.e., each revealed vertex is connected to the previously revealed vertex.

Note for $k = 0$, we have a special case where during the query it suffices to simply return a triangle τ that is below the lower envelope and above q. Also the restriction that $\mathbf{H}$ must be a walk for the on-demand version is not strictly necessary in RAM models. However, removing this restriction creates some technical issues in some variants of the pointer machine model and thus it is added to avoid needless complications. Furthermore, a walk is sufficient in all the applications that we will consider. Our main technical result is the following.

▶ **Theorem 5.** *The on-demand LEIS3D problem for an input of size n on catalog graph $\mathbf{G}$ can be solved with a data structure that uses $O(n)$ space and has the following query time: (i) $O(\log n \log \log n + |\mathbf{H}| \log \log n)$ if $\mathbf{G}$ is a path and (ii) $O(\log n + |\mathbf{H}| \sqrt{\log n})$ when $\mathbf{G}$ is a graph of constant degree.*

3 Building the Tools

In this section, we will prove a number of auxiliary results that will later be combined to build our main results.

3.1 Point Location in 3D Convex Overlays

The following observation is our starting point. The proof is not too difficult and is available in the full version [5].

▶ **Lemma 6.** *Let $\mathcal{P}_1, \cdots, \mathcal{P}_m$ be m convex polytopes in $\mathbb{R}^3$ of total complexity n. Let $\mathcal{A}$ be their overlay. The overlay $\mathcal{A}$ has $O(nm^2)$ complexity.*

It is critical to observe that an equivalent lemma for 2D subdivision does not hold. For example, the overlay of two subdivisions of the plane into n "tall and thin" and "wide and narrow" slabs has $\Omega(n^2)$ size. Thus, the above lemma captures a fundamental impact of adding convexity to the picture. The next lemma shows that we can perform point location in 3D via a simple idea and a small query overhead.

▶ **Lemma 7** (basic point location). *Let $S = \{\mathcal{P}_1, \cdots, \mathcal{P}_m\}$ be a set of m convex polytopes in $\mathbb{R}^3$ of total complexity n. We can build a data structure that uses $O(nm^2)$ space that does the following: The data structure stores a set T of "anchors" (triangles) such that each anchor is either inside a polytope or lies completely outside it. Then, given a query point q, it can output an anchor $\tau \in T$ such that $\Diamond(S, q)$ is equal to the subset of polytopes that contain τ. The time to answer the query is $O(\log n \log m)$.*

Proof sketch (see the full version [5] for the full proof). The plan is to do a binary search via point location queries. To do that, we decompose each polytope into an upper and lower hull and then turn each hull into an everywhere defined surface (a function from $\mathbb{R}^2$ to $\mathbb{R}$). We overlay them, triangulate them, and then decompose them into "levels" where each level ends up being a Z-monotone surface whose projection will be a planar decomposition which will be stored in a point location data structure. The level (and the triangle) immediately above a query point can be found with $O(\log m)$ steps of a binary search process that uses the point location queries for a total query time of $O(\log n \log m)$.   ◀

For some applications, the above lemma is sufficient but for our most general result, we need to remove the factor $\log m$ in the above query time.

▶ **Lemma 8** (advanced point location). *Let $S = \{\mathcal{P}_1, \cdots, \mathcal{P}_m\}$ be a set of m convex polytopes in $\mathbb{R}^3$ of total complexity n, where $m = O(2^{\sqrt{\log n}})$. We can build a data structure that uses $O(n2^{O(\sqrt{\log n})})$ space that does the following: The data structure stores a set T of "anchors" (triangles) such that each anchor is either inside a polytope or lies completely outside it. Then, given a query point q, it can output an anchor $\tau \in T$ such that $\Diamond(S, q)$ is equal to the subset of polytopes that contain τ. The query time is $O(\log n)$.*

The rest of this subsection is devoted to the proof of Lemma 8.

3.1.1   Preliminaries

A modified Dobkin-Kirkpatrick hierarchy.   First, we describe a process of simplifying polytopes in 3D which is a modified version of building Dobkin-Kirkpartrick hierarchies [16] but in a randomized way. We refer to the standard Dobkin-Kirkpartrick simplification as DK-simplification and to our modified randomized simplification as RDK-simplification. Let $\mathcal{P}$ be a convex polytope in 3D. An *outer simplification* of $\mathcal{P}$ is obtained by considering $\mathcal{P}$ as the intersection of halfspaces that bound its faces. Then, we select a subset I of independent faces of $\mathcal{P}$ (i.e., no two adjacent faces are selected) and such that each face has $O(1)$ complexity and $|I| = \Omega(|\mathcal{P}|)$. In DK-simplification, I is simply deleted, instead in RDK-simplification, we delete each element of I with probability 0.5, independently. Observe that the deletion of I' is equivalent to attaching $|I'|$ disjoint convex cells of constant complexity to the faces of $\mathcal{P}$. To define an *inner simplification* of $\mathcal{P}$ we do a similar process but this time we consider $\mathcal{P}$ as the convex of hull of its vertices and select an independent set I of the vertices of $\mathcal{P}$ of size $\Omega(|\mathcal{P}|)$ such that each vertex has constant degree. In the standard DK-simplification, I is simply deleted, but in RDK-simplification, we delete each vertex in I with probability 0.5, independently. This creates a convex polytope that is contained inside $\mathcal{P}$ (or shares some boundaries). This process can be thought of as removing disjoint convex cells of constant complexity from $\mathcal{P}$. See Figure 2 for an illustration.

We can repeatedly apply DK- or RDK-simplifications until the inner polytope is reduced to a simplex and the outer polytope to a halfspace. This results in a hierarchy which is denoted by $\mathcal{H}(\mathcal{P})$. An easy observation that we will be using is the following.
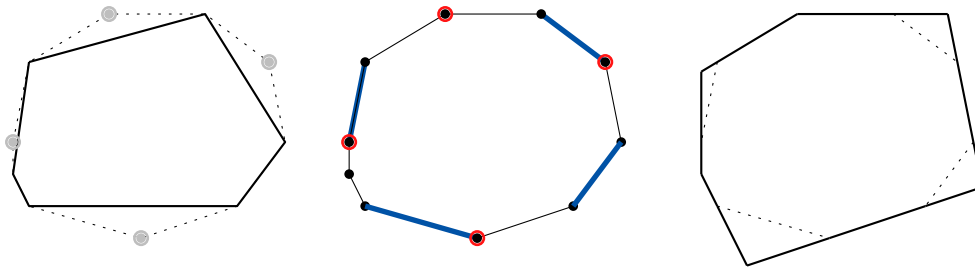
Figure 2 The inner and outer simplifications are shown in 2D, for clarity. The central polytope is being simplified here. To the left, we have its inner simplification after deleting the vertices marked with red and to the right, we have the outer simplification after deleting the edges marked with blue.

▶ **Observation 9.** *If every simplification reduces the complexity of $\mathcal{P}$ by a constant factor, then, $\mathcal{H}(\mathcal{P})$ consists of $O(\log n)$ convex polytopes, i.e., it is the overlay of $O(\log n)$ convex polytopes which decomposes $\mathbb{R}^3$ into $O(|\mathcal{P}|)$ convex cells of constant complexity.*

A simplification round. Let T be a parameter that will be defined later (T will actually be set to $\sqrt{\log n}$). Given a convex polytope $\mathcal{P}$, *one round of simplification of $\mathcal{P}$* is defined as follows. If the complexity of $\mathcal{P}$ is at most 2^T, then $\mathcal{P}$ is simplified via DK-simplification, reducing the complexity of $\mathcal{P}$ by a constant factor at each simplification to yield a hierarchy of $O(\log|\mathcal{P}|) = O(T)$ simplifications. Otherwise, starting from $\mathcal{P}$, we repeatedly apply inner (resp. outer) RDK-simplification T times to obtain T progressively smaller (resp. larger) polytopes. The smallest and the largest polytopes that are obtained are called the *boundary* polytopes. Based on this, we perform the following steps for every polytope $\mathcal{P}_i$ in the input.
1. Initialize a list $X = \{\mathcal{P}_i\}$ and the set $S = \{\mathcal{P}_i\}$ of simplifications of $\mathcal{P}_i$. Each element of S will maintain a *counter* that counts how many rounds it has been simplified for, and for $\mathcal{P}_i$ this counter is initialized to zero.
2. While X is not empty:
   a. Remove the first polytope $\mathcal{P}$ from X and let c be the counter of $\mathcal{P}$.
   b. Simplify $\mathcal{P}$ for one round, resulting in $O(T)$ simplifications which are added to S; all of these simplifications have their counter set to $c + 1$.
   c. If any boundary polytope is produced (i.e., if $\mathcal{P}$ had complexity at least 2^T), then they are added to X.

We also define a subdivision $\mathcal{A}_j$ is as the overlay of all the polytopes (over all sets S) whose counter is at least j. The following lemma captures some of the important properties of our construction.

▶ **Lemma 10.** *Assuming $T \geq c + \log\log n$ for a large enough constant c, the following properties hold with probability at least $1 - n^2$ (whp).*
   (I) *Let x be the maximum value of the counters of the polytopes in $\mathcal{A}_0$. We have $x = O\left(\frac{\log n}{T}\right)$ (whp).*
   (II) *Let M be the number of polytopes in $\mathcal{A}_0$. We have $M = O(m2^x T)$.*
   (III) *The size of $\mathcal{A}_j$ is $O(nM^2)$, for all j.*
   (IV) *$\mathcal{A}_j$ decomposes $\mathbb{R}^3$ into convex cells of complexity $O(M)$.*
   (V) *Let Δ be a cell in $\mathcal{A}_j$ and let $\mathcal{P}$ be a polytope in $\mathcal{A}_{j-1}$. Then, the number of vertices of $\mathcal{P}$ that are contained in Δ as well as the number of edges of $\mathcal{P}$ that intersect Δ is bounded by $O(2^{O(T)}\log n) = O\left(2^{O(T)}\right)$. (whp).*
   (VI) *The size of $\mathcal{A}_x$ is $O\left(\left(m2^x T2^T\right)^{O(1)}\right)$.*

Proof. For claim (I), observe that if a polytope $\mathcal{P} \in X$ has size 2^T, then it is simplified with a standard DK-simplification and it adds no additional polytopes to X. Thus, it suffices to consider polytopes that have complexity at least $2^T \geq \log^c n$. Now the claim (I) follows from a standard Chernoff bound: Since in our RDK-simplification, we will select an independent set I of size $\Omega(|\mathcal{P}|) = \Omega(c \log n)$ and on expectation, we will delete half of elements but since $\mathbb{E}[|I|] \geq c \log n$, it follows that with with high probability at least $|I|/4$ of the elements of I are deleted. This reduces the complexity of polytope by a constant fraction. Consequently, the number of RDK-simplifications steps an input polytope can go through is bounded by $O(\log n)$ and thus the number of rounds is $O(\frac{\log n}{T})$ which implies claim (I).

Claim (II) follows from the observation that during each round, each polytope creates two additional polytopes (whose counters are incremented by one). In other words, the number of polytopes of counter $c + 1$ is at most double the number of polytopes with counter c. Combined with (I) this implies that the total number of polytopes ever added to X is $O(m2^x)$. For each such a polytope (whose counter is at least j) the overlay $\mathcal{A}_j$ includes T additional polytopes in S.

Claim (III) is a consequence of Lemma 6 and (II).

To see (IV), consider one polytope $\mathcal{P}$ in $\mathcal{A}_j$. Observe that all the inner and outer simplifications of $\mathcal{P}$ are also included in $\mathcal{A}_j$ this is because our simplification process continues until the inner simplification of $\mathcal{P}$ is reduced to a simplex and the outer simplification of $\mathcal{P}$ is reduced to a halfspace. All of these simplifications have a counter value that is at least j and thus by definition of $\mathcal{A}_j$ they are included in $\mathcal{A}_j$. Thus, some DK-hierarchy, $\mathcal{H}(\mathcal{P})$, of $\mathcal{P}$ is included in $\mathcal{A}_j$. By Observation 9, $\mathcal{H}(\mathcal{P})$ decomposes $\mathbb{R}^3$ into cells of constant complexity. The number of such hierarchies that are included in $\mathcal{A}_j$ is upper bounded by the number of polytopes, M. Each cell Δ of $\mathcal{A}_j$ is thus the intersection of at most M convex cells of constant complexity and thus Δ will have complexity at most $O(M)$.

Now consider claim (V). Observe that if $\mathcal{P}$ is included in $\mathcal{A}_j$, then Δ cannot contain any vertex of $\mathcal{P}$ or intersect any of its edges. Thus, we can assume $\mathcal{P}$ is not included in $\mathcal{A}_j$. But this also implies that the counter of $\mathcal{P}$ must have value exactly $j - 1$. At some point $\mathcal{P}$ will be considered during our simplification process. Assume that Δ is intersected by Y edges of $\mathcal{P}$ and $Y = \Omega(2^T \log n)$. This implies that $\mathcal{P}$ must have undergone our RDK-simplification process. Consider one step of our randomized DK-simplification: regardless of whether we are dealing with inner or outer simplification and regardless of which independent set is chosen, each of these Y edges are kept with probability at least $\frac{1}{4}$. By a standard Chernoff bound, it follows that with high probability, at least $\frac{1}{8}$ fraction of them are kept. $\mathcal{P}$ undergoes at most T such simplification steps until its counter is incremented by one and the resulting boundary polytopes are added to $\mathcal{A}_j$. This implies that with high probability, the resulting boundary polytopes have at least $\frac{Y}{8^T}$ edges intersecting Δ. If $Y \geq c8^T \log n$, this is a contradiction and thus the claim follows. A similar argument applies to the vertices of $\mathcal{P}$ contained in Δ.

Finally, the claim (VI) follows by the observation that at the last round, all polytopes must have complexity at most 2^T and there can be at most $m2^x T$ of them by claim (II) and now we plug these values in Lemma 6 to prove this claim. ◀

The next lemma enables us to give an upper bound on the number of cells of $\mathcal{A}_{j-1}$ that can intersect the cells of $\mathcal{A}_j$. This is a very crucial part of our point location data structure but the proof is relatively technical so it is postponed to the full version [5].

▶ **Lemma 11.** *Let A and B be the overlay of at most L convex polytopes each such that they form a decomposition of $\mathbb{R}^3$ into convex cells of complexity at most δ. In addition, assume that for every cell Δ in B, and every polytope $\mathcal{P}$ in A, the number of vertices of $\mathcal{P}$ that are contained in Δ and the number of edges of $\mathcal{P}$ that intersect Δ are bounded by X.*

Then, every cell Δ of B is intersected by at most $O(L^3X + L^3\delta + L^2\delta^2 + L\delta^3)$ cells of A.

3.1.2 Proof of Lemma 8 (Advanced Point Location)

Our data structure is as follows. Set $T = \sqrt{\log n}$. Observe that we have $2^T = 2^{\frac{\log n}{T}}$. We build the previously mentioned overlays $\mathcal{A}_i$ for $i = 1, \cdots, O(\log_T n)$. Then, for each cell Δ in $\mathcal{A}_j$, we find all the cells Δ' in $\mathcal{A}_{j-1}$ that intersect Δ and then store the hyperplanes that define them in what we call a *fast data structure*. Given a set of ℓ hyperplanes, such a fast data structure can store them using $O(\ell^3)$ space such that point location queries can be answered in $O(\log \ell)$ time [12]. This concludes our data structure. Our query algorithm is relatively simple: if we know the cell Δ in $\mathcal{A}_j$ that contains the query point, then by querying the fast data structure built for Δ, we can find the cell Δ' in $\mathcal{A}_{j-1}$ that contains the query point and then we can continue until we finally point locate the query. We now bound the space and the query time of the data structure.

The query time. By Lemma 10 (VI), we can locate the query point q in a cell of $\mathcal{A}_x$ in $O(\log m + x + \log T + T) = O(\sqrt{\log n})$ time. Thus, assume, we know q is in a cell Δ of $\mathcal{A}_j$. By Lemma 10, each $\mathcal{A}_j$ will have total complexity $O(n2^{O(T)})$ and is composed of $2^{O(T)}$ polytopes with cells of complexity $2^{O(T)}$. By Lemma 11 (using $\delta, L = M, X$ all set to $2^{O(T)}$) and Lemma 10(II,IV,V), Δ will intersect at most $2^{O(T)}$ cells Δ', whp. Note that we can assume this holds in the worst-case because if a cell intersects more than this amount, then we can simply rebuild the data structure and the expected number of such trials is $O(1)$. Thus, querying the fast data structure will take $O(T)$ time. Finally, observe that the depth of the recursion is $x = \frac{\log n}{T}$. So we get the query time of $O(\log n)$.

The space analysis. As mentioned each $\mathcal{A}_i$ will have complexity $O(n2^{O(T)})$. For each cell we store a fast data structure that uses $2^{O(T)}$ space, leading to $O(n2^{O(T)})$ total space usage.

3.2 Solving LEIS3D

In this section, we prove our main technical tool.

▶ **Theorem 5.** *The on-demand LEIS3D problem for an input of size n on catalog graph $\mathbf{G}$ can be solved with a data structure that uses $O(n)$ space and has the following query time: (i) $O(\log n \log \log n + |\mathbf{H}| \log \log n)$ if $\mathbf{G}$ is a path and (ii) $O(\log n + |\mathbf{H}|\sqrt{\log n})$ when $\mathbf{G}$ is a graph of constant degree.*

Proof. Note that when $\mathbf{G}$ is a general graph of maximum degree $d = O(1)$, we can assume that its maximum degree is 3 by replacing a vertex of degree $d' \le d$ with a binary tree of depth $\log d'$ which only blows up the query time by a constant factor.

Recall that our data structure should store a k-shallow cutting of H_v for each vertex $v \in \mathbf{G}$, and let c be a large enough constant. When $\mathbf{G}$ is a path we set $\ell = \log n \cdot \log \log n$ and $x = k(2\ell)^c$ and when $\mathbf{G}$ is a general graph, we set $\ell = \sqrt{\log n}$ and $x = k2^{c\ell}$. We say a vertex $u \in \mathbf{G}$ is *light* if $|H_v| \le x$, and *heavy* otherwise. For a vertex v, its ℓ-*neighborhood*, denoted by $\mathcal{N}_\ell(v)$, is the set of all the vertices of $\mathbf{G}$ that have distance at most ℓ to v.

The data structure. For every vertex v, we build a x-shallow cutting $\mathcal{C}_v$ for the set of hyperplanes H_v. If v is heavy, $\mathcal{C}_v$ is built via Lemma 3 (together with the all the corresponding apparatus in the lemma) and thus $\mathcal{C}_v$ is assumed to be convex (i.e., the lower envelope of some hyperplanes). However, if v is light, then $\mathcal{C}_v$ is assumed to consist of just one triangle τ, the hyperplane $Z = +\infty$ as an infinite triangle and the conflict list of τ in this case is H_v. In either case, for every triangle τ in the shallow cutting $\mathcal{C}_v$, we compute and store the
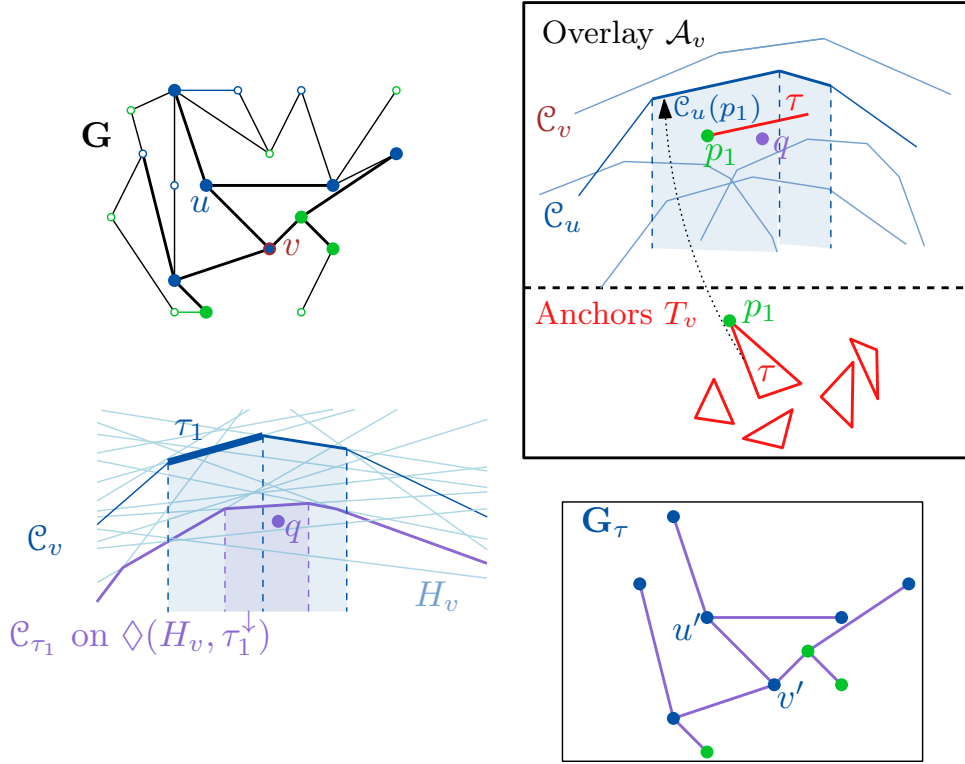
■ **Figure 3** An overview of the data structure. Light nodes shown in green, heavy nodes in blue, and $\mathcal{N}_2(v)$ is marked with large disks. Each triangle τ in the point location data structure on $\mathbf{G}_u$ stores a pointer to a copy of this neighborhood. From there, we can jump to cells in $\mathcal{C}_v$.

conflict list, $\Diamond(H_v, \tau^\downarrow)$, and then we build a k-shallow cutting, $\mathcal{C}_\tau$, on $\Diamond(H_v, \tau^\downarrow)$; we call $\mathcal{C}_\tau$ *the secondary shallow cutting*. This is also built via Lemma 3. Note that as $x \geq k$, these secondary shallow cuttings cover the $\leq k$-level of H_v.

The overlays. Next, we consider each heavy vertex v: we consider all the vertices $u \in \mathcal{N}_\ell(v)$ and the shallow cuttings $\mathcal{C}_u$ as convex polytopes and create a 3D point location data structure on the overlay all the convex polytopes $\mathcal{C}_u$, for all u. Let $\mathcal{A}_v$ be this overlay. When $\mathbf{G}$ is a path, $\mathcal{N}_\ell(v)$ contains at most 2ℓ vertices and this overlay is built via Lemma 7. When $\mathbf{G}$ is a graph, it contains at most $2^{\ell+1}$ vertices and we use Lemma 8. In either case, the data structure will store a number of triangles as anchors, denoted by T_v. For each anchor $\tau \in T_v$ we consider its corners (vertices) p_1, p_2 and p_3. Then, consider every $u \in \mathcal{N}_\ell(v)$ and its shallow cutting $\mathcal{C}_u$; store a pointer from τ to the triangle, $\mathcal{C}_u(p_i)$, in $\mathcal{C}_u$ that lies above p_i, for $1 \leq i \leq 3$. Observe that for the anchor τ, we store at most $3\mathcal{N}_\ell(v)$ pointers (at most three per $u \in \mathcal{N}_\ell(v)$). These pointers are arranged in a particular way: we make a copy of $\mathcal{N}_\ell(v)$, denoted by $\mathbf{G}_\tau$, where each vertex $u \in \mathcal{N}_\ell(v)$ has a copy $u' \in \mathbf{G}_\tau$ and we store the three pointers from u' to $\mathcal{C}_u(p_i)$, for $1 \leq i \leq 3$.

The query. Consider a query, consisting of a point $q \in \mathbb{R}^3$ and an initial vertex u of the query subgraph $\mathbf{H}$.

1. If u is light, $\mathcal{C}_u$ contains one triangle τ which is the plane $Z = +\infty$ and one cell which is the entire $\mathbb{R}^3$. We look at the secondary shallow cutting $\mathcal{C}_\tau$ and locate $\mathcal{C}_\tau(q)$ if it exists and return conflict list $\Diamond(H_u, \mathcal{C}_\tau(q)^\downarrow)$ which by Lemma 3 contains $\Diamond(H_u, q^\downarrow)$.

2. This process can be repeated until we encounter the first heavy vertex u. We query the 3D point location data structure (Lemma 8 for general graphs and Lemma 7 for when $\mathbf{G}$ is a path) to get an anchor triangle τ with corners p_1, p_2 and p_3. We use the pointers associated with the copy u' of u in $\mathbf{G}_\tau$ to find the triangle $\tau_i = \mathcal{C}_u(p_i)$, $1 \le i \le 3$. The 3D point location data structure guarantees that the corners of τ all lie below $\mathcal{C}_u$, and thus all three of these triangles τ_i exist, if and only if q lies below $\mathcal{C}_u$.

   We query the secondary shallow cuttings $\mathcal{C}_{\tau_i}$, $1 \le i \le 3$, and for each, we find $\mathcal{C}_{\tau_i}(q)$, if it exists and then report the pointer to the corresponding secondary conflict list.

3. Now assume the query is given a sequence of vertices of $\mathbf{G}$, $u_0 = u, u_1, \cdots, u_m$, one by one, such that they all belong to $\mathcal{N}_\ell(u)$. Let τ be the triangle found in the previous step. The crucial observation is that τ does not intersect any triangle in the shallow cuttings $\mathcal{C}_w$, for any $w \in \mathcal{N}_\ell(u)$ since τ was a triangle in the overlay of all the shallow cuttings. Thus, in this case, the anchor does not change. In this case, for each revealed vertex u_j, we simply need to follow the pointer from u'_{j-1} to u'_j in $\mathbf{G}_{u,\tau}$ and from u'_j to $\mathcal{C}_{u_j}(p_i)$, for $1 \le i \le 3$. The rest of the algorithm is the same as Step 2.

4. Eventually, the algorithm will reveal a vertex u_{m+1} that is outside the $\mathcal{N}_\ell(u)$. In this case, we simply go to Step 1 or 2, i.e., we assume u_{m+1} is the initial vertex.

The query time. Step 1 takes $O(\log(\frac{x}{k}))$ time by Lemma 3 since each conflict in $\mathcal{C}_v$ has size $O(x)$ and we have a k-shallow cutting for them. In Step 2, we use $Q = O(\log n \log \log n)$ time for when $\mathbf{G}$ is a path by Lemma 7 and $Q = O(\log n)$ time for a general graph to identify the anchor τ. Obtaining the triangles $\mathcal{C}_u(p_i)$ for $1 \le i \le 3$ is a constant time operation and then we again query the secondary shallow cuttings which takes $O(\log(\frac{x}{k}))$ time. In Step 3, the query time is only $O(\log(\frac{x}{k}))$ because we do not need to find the anchor τ, and the cost is simply the cost of querying the secondary shallow cutting. Finally, observe that whenever we reach Step 2, at least ℓ other vertices of $\mathbf{H}$ must be revealed until we exceed $\mathcal{N}_\ell(u)$. In other words, Step 2 is only repeated once ℓ additional vertices of $\mathbf{H}$ have been revealed. Summing this up, we get that the query time is asymptotically bounded by

$$Q + \log\left(\frac{x}{k}\right)|\mathbf{H}| + \frac{|\mathbf{H}|}{\ell} \cdot Q.$$

By plugging in the corresponding parameters, when $\mathbf{G}$ is a path, the query time is

$$O(\log n \log \log n + |\mathbf{H}| \log \log n)$$

and for a general graph $\mathbf{G}$ it is

$$O(\log n + |\mathbf{H}|\sqrt{\log n}).$$

The space analysis. The total size of the shallow cuttings $\mathcal{C}_v$ and the secondary shallow cuttings is $O(n)$. Let M be the maximum size of $\mathcal{N}_\ell(v)$, i.e., for a path $\mathbf{G}$ we have $M = 2\ell$ and for a general graph of maximum degree three we have $M = 2^\ell$. Observe that the parameter x is at least M^c in both cases, for some constant c that we can choose. The number of heavy vertices is at most $\frac{n}{x}$ and each heavy vertex can appear in the ℓ-neighborhood of at most M other vertices. Thus, the total number of overlays that we will create is at most $\frac{nM}{x}$ and each overlay will contain at most M convex polytopes. By choosing c large enough and by plugging the bounds from Lemma 7 and Lemma 8, we get that the total space of the 3D point location can be bounded by $O\left(\frac{n}{M}\right)$ which is also an upper bound for the number of anchors. Finally, for every anchor, we store at most M pointers for a total of $O(n)$ space. ◀

4 Some Applications

Here, we briefly mention quick applications of Theorem 5. Note that by lifting map, similar problems for 2D disks can also be solved.

▶ **Theorem 12.** *Let P be a set of n points in 3D, each associated with a real-valued weight. We can store P in a data structure that uses $O(n \log n)$ space such that given a query halfspace h and two values w_1 and w_2, we can find all the points P that lie inside h and whose weight lies between w_1 and w_2 in $O(\log^{3/2} n + t)$ query time where t is the size of the output.*

Proof. Use duality to get a set of n hyperplanes and let q be point dual to the query halfspace h. Store the hyperplanes in a balanced binary tree **G**, ordered by weight. Every node v of **G** defines a *canonical set H_v* which is the subset of hyperplanes stored in the subtree of v. Store H_v in a data structure of Afshani and Chan [2]. This uses $O(n \log n)$ space in total. Then, answering the query can be reduced to answering $O(\log n)$ 3D halfspace range reporting queries on two root to leaf paths in the tree. Simply querying the data structure of [2] gives $O(\log^2 n + t)$ query time which is already optimal if $t \geq \log^2 n$. To improve the query time, it thus suffices to assume $t \leq \log^2 n$. In which case, we use Theorem 5 with parameter $k = \log^2 n$ and store each conflict list stored by Theorem 5 in a data structure of [2]. This combination still uses linear space and when $t \leq \log^2 n$, it returns $O(1)$ conflict lists of size $O(\log^2 n)$ each. Querying the data structure of [2] on the conflist lists gives the claimed query time. ◀

Note that offline halfspace range reporting queries can also be solved with a very similar technique. We now consider 3D halfspace max queries.

▶ **Theorem 13.** *3D halfspace range max queries can be solved $O(n \log n)$ space and $O(\log^{3/2} n)$ query time.*

Proof. Use duality and let $H = \{h_1, .., h_n\}$ be the dual input hyperplanes ordered by increasing weights, and let w_i be the weight of h_i. We construct a balanced binary tree **G**, whose leaves correspond to the hyperplanes $h_1, .., h_n$. For each node v, let H_v denote the subset of hyperplanes stored in the leaves below v. We now build the data structure of Theorem 5, with parameter $k = 0$. Total number of hyperplanes is $O(n \log n)$, the data structure uses $O(n \log n)$ space.

Consider the query point $q \in \mathbb{R}^3$ dual to the query halfspace, and let h_i be the (unknown) heaviest plane that passes below q. To find h_i, we construct a path **H** of length $O(\log n)$ in an on-demand manner. The main idea is as follows. At node v, with left child ℓ and right child r, we want to test if q lies below the lower envelope (the 0-level) of H_r. If so, then h_i must lie in the left subtree rooted at ℓ, hence we continue the search there. If q lies above the lower envelope of H_r, we continue the search in the subtree rooted at r. The process finishes after we find the leaf containing h_i in $O(\log n)$ rounds. This process generates a path **H** of length $O(\log n)$. Hence, the total query time is $O(\log^{3/2} n)$ as claimed. ◀

As discussed, this also improves the query time of the best range sampling data structures for weighted points. See [6]. It can also be combined with Theorem 12 to answer "top-k" reporting queries of halfspaces in 3D [19] as well as approximate counting queries [18].

▶ **Theorem 14.** *Let P be a set of points, each associated with a color from the set $[m]$. We can store P in a data structure, s.t., given a query halfspace h, we can report the t distinct colors of the points in h in (i) $O(\log n \log \log + m \log \log n)$ time using $O(n)$ space or (ii) $O(\log n + t \log(m/t)\sqrt{\log n})$ time using $O(n \log m)$ space.*

Proof. Once again use duality and let H be the of dual halfspaces. Let H_i be the hyperplanes of color i, $1 \le i \le m$. For the first claim (i), simply use a path of length m as graph $\mathbf{G}$ and invoke Theorem 5 where all query subgraphs $\mathbf{H}$ are equal to $\mathbf{G}$.

For the second claim (ii), add a balanced binary tree of height $\log m$ on the path and let $\mathbf{G}$ be the resulting graph. For every node $v \in \mathbf{G}$, define H_v to be the union of hyperplanes stored in the subtree of v. We now build the Theorem 5 data structure with $k = 0$ the graph $\mathbf{G}$. The total size of all sets H_v, and thus of our data structure is $O(n \log m)$. Now, consider the query point $q \in \mathbb{R}^3$ dual to h. Once again, we use the on-demand capability and at every node $v \in \mathbf{G}$ we decide whether either child of v has any output color, and if so, we recurse into that child. Let $\mathbf{H}$ denote the subtree of $\mathbf{G}$ visited by this procedure. By classical results, size of $\mathbf{H}$ is $O(t \log(n/t))$, when t is the number of leaves visited (and thus the number of distinct colors reported). Note that $\mathbf{H}$ can be assumed to be a walk by simpling fully finishing the recursion on each child of v, then backing up to v and then potentially recursing on the other child. The total query time is $O(\log n + t \log(m/t)\sqrt{\log n})$. ◀

References

1 Peyman Afshani and Timothy M Chan. On approximate range counting and depth. In *Proceedings of the twenty-third annual symposium on Computational geometry*, pages 337–343, 2007. `doi:10.1145/1247069.1247129`.

2 Peyman Afshani and Timothy M. Chan. Optimal halfspace range reporting in three dimensions. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 180–186, 2009. `doi:10.1137/1.9781611973068.21`.

3 Peyman Afshani and Pingan Cheng. 2D generalization of fractional cascading on axis-aligned planar subdivisions. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 716–727, 2020. `doi:10.1109/FOCS46700.2020.00072`.

4 Peyman Afshani, Pingan Cheng, Aniket Basu Roy, and Zhewei Wei. On range summary queries. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 7–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, 2023.

5 Peyman Afshani, Yakov Nekrich, and Frank Staals. Convexity Helps Iterated Search in 3D. *CoRR*, 2504.07545, 2025. `arXiv:2504.07545`.

6 Peyman Afshani and Jeff M. Phillips. Independent Range Sampling, Revisited Again. In *Symposium on Computational Geometry (SoCG)*, pages 4:1–4:13, 2019. `doi:10.4230/LIPICS.SOCG.2019.4`.

7 Peyman Afshani, Bryan Wilkinson, Yufei Tao, and Cheng Sheng. Concurrent range reporting in two-dimensional space. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 983–994, 2014. `doi:10.1137/1.9781611973402`.

8 Boris Aronov and Sariel Har-Peled. On approximating the depth and related problems. *SIAM Journal on Computing*, 38(3):899–921, 2008. `doi:10.1137/060669474`.

9 Timothy M. Chan. Random sampling, halfspace range reporting, and construction of ($\le$ k)-levels in three dimensions. *SIAM Journal of Computing*, 30(2):561–575, 2000. `doi:10.1137/S0097539798349188`.

10 Timothy M. Chan and K. Tsakalidis. Optimal deterministic algorithms for 2-d and 3-d shallow cuttings. *Discrete & Computational Geometry*, 56, 2016.

11 Timothy M. Chan and Da Wei Zheng. Hopcroft's problem, log-star shaving, 2D fractional cascading, and decision trees. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 190–210, 2022. `doi:10.1137/1.9781611977073.10`.

12 Bernard Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete & Computational Geometry*, 9(2):145–158, 1993. `doi:10.1007/BF02189314`.

13 Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986. `doi:10.1007/BF01840440`.

14 Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986. `doi:10.1007/BF01840441`.

15 Bernard Chazelle and Ding Liu. Lower bounds for intersection searching and fractional cascading in higher dimension. *Journal of Computer and System Sciences*, 68(2):269–284, 2004. `doi:10.1016/J.JCSS.2003.07.003`.

16 David P Dobkin and David G Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. *Journal of Algorithms*, 6(3):381–392, 1985. `doi:10.1016/0196-6774(85)90007-0`.

17 Sariel Har-Peled and Micha Sharir. Relative (p, ε)-approximations in geometry. *Discrete & Computational Geometry*, 45(3):462–496, 2011. `doi:10.1007/S00454-010-9248-1`.

18 Saladi Rahul. Approximate range counting revisited. In *Symposium on Computational Geometry (SoCG)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.

19 Saladi Rahul and Yufei Tao. Generic techniques for building top-k structures. *ACM Transactions on Algorithms*, 18(4):38:1–38:23, 2022. `doi:10.1145/3546074`.

20 Edgar A. Ramos. On range reporting, ray shooting and k-level construction. In *Symposium on Computational Geometry (SoCG)*, pages 390–399, 1999. `doi:10.1145/304893.304993`.

21 Jiří Matoušek. Reporting points in halfspaces. *Computational Geometry: Algorithms and Applications*, 2(3):169–186, 1992. `doi:10.1016/0925-7721(92)90006-E`.

22 V.K Vaishnavi and D Wood. Rectilinear line segment intersection, layered segment trees, and dynamization. *Journal of Algorithms*, 3(2):160–176, 1982. `doi:10.1016/0196-6774(82)90016-5`.

23 Dan E. Willard. New data structures for orthogonal range queries. *SIAM Journal of Computing*, 14:232–253, 1985. `doi:10.1137/0214019`.