Approximate Nearest Neighbor Search in Metrics of Planar Graphs

Ittai Abraham[1], Shiri Chechik[2], Robert Krauthgamer[*3], and Udi Wieder[3]

1 **VMware Research, Palo Alto, CA, USA,** `{iabraham,uwieder}@vmware.com`
2 **Tel-Aviv University, Tel-Aviv, Israel,** `shiri.chechik@gmail.com`
3 **Weizmann Institute of Science, Israel,** `robert.krauthgamer@weizmann.ac.il`

Abstract

We investigate the problem of approximate Nearest-Neighbor Search (NNS) in graphical metrics: The task is to preprocess an edge-weighted graph $G = (V, E)$ on m vertices and a small "dataset" $D \subset V$ of size $n \ll m$, so that given a query point $q \in V$, one can quickly approximate $\mathbf{d_G}(q, D)$ (the distance from q to its closest vertex in D) and find a vertex $a \in D$ within this approximated distance. We assume the query algorithm has access to a distance oracle, that quickly evaluates the exact distance between any pair of vertices.

For planar graphs G with maximum degree Δ, we show how to efficiently construct a compact data structure – of size $\tilde{O}(n(\Delta + 1/\epsilon))$ – that answers $(1 + \epsilon)$-NNS queries in time $\tilde{O}(\Delta + 1/\epsilon)$. Thus, as far as NNS applications are concerned, metrics derived from bounded-degree planar graphs behave as low-dimensional metrics, even though planar metrics do not necessarily have a low doubling dimension, nor can they be embedded with low distortion into ℓ_2. We complement our algorithmic result by lower bounds showing that the access to an exact distance oracle (rather than an approximate one) and the dependency on Δ (in query time) are both essential.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases Data Structures, Nearest Neighbor Search, Planar Graphs, Planar Metrics, Planar Separator

Digital Object Identifier 10.4230/LIPIcs.APPROX-RANDOM.2015.20

1 Introduction

In the Nearest Neighbor Search (NNS) problem, the input is a dataset $D = \{p_1, \ldots, p_n\}$ containing n points that lie in some large host metric space (M, d). These points should be preprocessed into a data structure so that given a query point $q \in M$, the dataset point $p_i \in D$ closest to q can be reported quickly. The scheme's efficiency is typically measured by the space complexity of the data structure and the time complexity of the query algorithm. NNS is a fundamental problem with numerous applications, and has therefore attracted a lot of attention, including extensive experimental and theoretical analyses. Often, finding the exact closest neighbor is relaxed to finding an approximate solution, called $(1+\epsilon)$-NNS, where the goal is to report a dataset point $p_i \in D$ satisfying $d(q, p_i) \leq (1+\epsilon) \min\{d(q, p_j) \mid p_j \in D\}$.

Previous work on NNS has largely focused on the case where the host metric is ℓ_p-norm for some p, typically ℓ_1 or ℓ_2, over $M = \mathbb{R}^m$ for some dimension $m > 0$. In this common setting, exact NNS exhibits a "curse of dimensionality" – either the space (storage requirement) or

* Work supported in part by a US-Israel BSF grant #2010418 and an Israel Science Foundation grant #897/13.

18th Int'l Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX'15) /
19th Int'l Workshop on Randomization and Computation (RANDOM'15).
Editors: Naveen Garg, Klaus Jansen, Anup Rao, and José D. P. Rolim; pp. 20–42
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the query time must be exponential in the dimension – which provides a strong motivation to study approximate solutions. When the dimension m is constant, known $(1+\epsilon)$-NNS algorithms achieve almost linear space and a polylogarithmic query time; but when the dimension is logarithmic (in n), all known algorithms, including approximate ones, require (in the worst-case) either a super-linear space or a polynomial query time.

While the ℓ_p-norm setting captures many applications, certain data types cannot be embedded (with low distortion) into $\mathbb{R}^m$, and it is therefore desirable to consider other metric spaces. However, the notion of dimensionality is not well-defined for general metrics, and it is unclear a-priori what type of internal structure suffices for efficient approximate NNS algorithms. A notable example of such an approach, initiated by [22, 17, 18], is the study of NNS in general metric spaces assuming the algorithm has access to a distance oracle, i.e., that the distance function can be evaluated in unit time; their motivation was drawn from the analysis of computer networks, where distances are derived from a huge graph (rather than by a norm or another simple function of the vertex names). It is known [18, 19, 8, 15, 12] that if the metric space restricted to the n dataset points has a bounded doubling dimension, then $(1+\epsilon)$-NNS can be solved with near-linear space and polylogarithmic query time.

1.1 Our Results

We look at another family of metric spaces, those derived by way of shortest paths from a graph with positive edge weights. Similarly to [22, 17, 18], we assume the NNS algorithm has access to a distance oracle.

As a motivating application consider the case where the graph represents a road network, say of the continental United States. Even though this graph has tens of millions of nodes, extremely efficient exact distance oracles have been built for it (see [14, 6, 2]). Now suppose we wish to find the nearest shop from a collection like the set of all Starbucks shops, which currently has roughly $12,000$ locations in the US. This means we want to design a compact application (e.g., mobile app) that has access to a generic server (like google maps). While the server could use much larger space (but cannot be customized to support specialized operations like NNS), the application must be very efficient in terms of query time and space, and thus our goal is to build an NNS data structure whose efficiency depends on the significantly smaller number of shops ($n = |D|$ in our notation).

Our main result is that in planar graphs of bounded degree, $(1+\epsilon)$-NNS can be solved using near-linear space and polylogarithmic query time. Thus, bounded-degree planar metrics exhibit "low-dimensional" behavior, even though they do not necessarily have a low doubling dimension, nor can they be embedded with low distortion into ℓ_2. This phenomenon, namely, that the restricted topology of planar graphs maintains some of the geometric structure of the Euclidean plane, is known in other contexts like compact routing and TSP, but here we show it for the first time in the context of NNS.

▶ **Theorem 1.** *Let (M,d) be a metric derived from a plane graph of maximum degree $\Delta > 0$ with positive edge weights, and let $\epsilon > 0$. Then every dataset $D \subset M$ of size $n = |D|$ can be preprocessed into a data structure of size $O(\epsilon^{-1} n \log n \log |M| + n\Delta \log^2 n)$ words, which can answer $(1+\epsilon)$-NNS queries in time $O((\epsilon^{-1} \log\log n + t_{DO}) \log n \log |M| + \log n \cdot \Delta t_{DO})$, assuming the distance between any two points in M can be computed in time t_{DO}.*

The data structure's size is measured in words, where a single word can accommodate a point in M or a (numerical) distance value. The term plane graph refers to a planar graph accompanied with a specific drawing in the plane.

Theorem 1 makes two assumptions, that there is a bound on the maximum degree, and that the data structure has access to an exact distance oracle. We further show (in Section 4) that both of these assumptions are necessary. Roughly speaking, we prove that if the degree is large, or if the distance oracle is approximate, the graph could contain symmetries that only a large number of accesses to the distance oracle could break.

▶ Remark. We cannot dispose of the maximum degree assumption by "vertex splitting", where a high-degree vertex is replaced with a binary tree with zero edge weights, because we assume access to a distance oracle for G (but not necessarily for a modified graph G'), since the distance oracle models a generic server. For the same reason, we cannot make the usual assumptions that G is triangulated or perturb the pairwise distances to be all distinct.

1.2 Related Work

Our model of NNS for a dataset D embedded inside a (huge) graph G is related to vertex-sparsification of distances [20], where the goal is to construct a small graph G' that (i) contains all the dataset point D (called terminals here) and furthermore maintains all their pairwise distances; and (ii) is isomorphic to a minor of G.

Here is another interesting related problem that is open: Given only the distances between a dataset D, find in polynomial time a *planar* host graph G that contains D and realizes their given pairwise distances (perhaps even approximately).

A key difference of our NNS model from these two problems is that the vertices outside of D are actually used explicitly as query points. However, one may hope for some connections, at least at a technical level.

1.3 Techniques

At a very high level, our algorithm is reminiscent of the classical k-d tree algorithm for NNS due to Bentley [7], as it partitions the graph recursively using separators that split the current dataset in an approximately balanced manner. While k-d trees use hyperplanes as separators of the host space, for planar metrics we use shortest-paths as separators (see [23, 3, 1, 13]). This recursive partitioning process can be described as creating a "hierarchy" tree $\mathcal{T}$, whose nodes correspond to "regions" in the metric space, and every leaf node represents a region with at most one dataset point; in our graphical case, every region is an induced subgraph of G. Given a query point q, one often uses a top-down algorithm to identify the leaf node in the hierarchy tree $\mathcal{T}$ that "contains" q, by tracing the location of q along the recursive partitioning, a process that we call the "zoom-in" phase. While this phase is trivial in k-d trees, and simple in bounded treewidth graphs (see Section 1.4), it is quite non-trivial in planar graphs, as explained below.

The key observation that completes the k-d tree algorithm is that once the tree leaf node containing q has been located, the nearest neighbor of q must lie "near the boundary" of one of the regions along the tree path leading to this leaf. Thus, all we need to store is just the separators themselves and the dataset points near them.

In planar graphs, this master plan has two serious technical difficulties. First, the separators themselves are too large to be stored explicitly. Recalling that the separators consist of shortest paths, we can employ the known trick [23, 3, 1, 13] of using a carefully chosen "net" to store them within reasonable accuracy, but since we actually need a net of all nearby dataset points, our solution is more involved and roughly uses a net of nets. Second, tracing the path to q along the hierarchy tree is a major technical challenge because our storage is proportional to $n = |D|$, while the separator size could be much larger, even linear in $|M|$. Our solution is to store just enough auxiliary information to identify at each level of the tree a few (rather than one) potential nodes, which suffices to “zoom-in” towards a small set of leaves that one of them contains q. The construction is presented in Section 2.

As a warm-up to the main result, we demonstrate our approach on the much simpler case of graphs with a bounded treewidth, where our algorithm solves NNS *exactly* using a standard tool of vertex separators of bounded size. This is only an initial example how NNS algorithms can leverage topological information, and the case of planar graphs is considerably more difficult — our algorithm uses a path separator, whose size is not bounded, and consequently solves NNS *approximately* (within factor $1 + \epsilon$).

1.4 Warmup: Bounded Treewidth Graphs

▶ **Theorem 2.** *Let D be a dataset of n points in a metric (V, d) derived from an edge-weighted graph of treewidth $w \geq 1$ and maximum degree $\Delta > 0$. Then D can be preprocessed into a data structure of size $O(\Delta w n)$ words, which can answer (exactly) nearest neighbor search queries in time $O(\Delta w \log n \cdot t_{DO})$, assuming the distance between any two points in V can be computed in t_{DO} steps.*

We assume for simplicity of exposition that an optimal tree decomposition is given to us; otherwise, it is possible to compute in polynomial time a tree decomposition of width $O(w \log w)$ [5] (or for fixed w, one of the algorithms of [9, 10]).

We use the following well-known property of bounded treewidth graphs: Given a set $X \subset V$, we can efficiently find a *separator* $S \subset V$ of size $|S| \leq w + 1$ whose removal breaks G into connected components $V_1, V_2, \ldots$ such that $|V_i \cap X| \leq |X|/2$ for all i. (The separator can be found by picking a single suitable node in the tree decomposition, and the width bound implies the bound on the separator size, see e.g. [11, Lemma 6].) It follows that in G, every path from V_i to V_j for $i \neq j$, must intersect the separator S.

The preprocessing phase

Given a dataset $D = \{p_1, \ldots, p_n\}$, recursively compute a partition (using the above property) with respect to the dataset points in the current component (i.e., $D \cap V'$ where $V' \subseteq V$ denotes the current component), until no dataset points are left (they were all absorbed in the separators). It is easy to see that the depth of the recursion tree is $O(\log n)$, and the number of separators used (non-leaf nodes in the recursion tree) is at most $O(n)$, each with at most $w + 1$ nodes. The data structure stores all the separators explicitly arranged in the form of their recursion tree. In addition, for each vertex u in any of these separators, it stores the following meta-data:

- All the neighbors (at most Δ) of u, along with the index of the part V_i to which they belong.
- A dataset point that is closest to u, i.e., $\operatorname{argmin}_{p \in D} d(p, u)$, breaking ties arbitrarily.

The total number of vertices in all the separators is at most $O(nw)$ and the meta-data held for every separator vertex u is of size $O(\Delta)$, hence the total size is $O(\Delta w n)$ words.

The query phase

We first argue that given a query point $q \in V$ and a separator S, it is possible to check, using the meta-data stored in the preprocessing phase, which part V_i contains q. This would imply that we can trace the path along the recursion tree all the way down to the last component containing q, a process that was mentioned before as the “zoom-in” phase. Indeed, finding this V_i is done by finding a vertex $u \in S$ that is closest to q, this task takes at most $|S| = w+1$ distance queries. If $u = q$ we are done. Otherwise, compute u's neighbor that is closest to q, namely, $v = \text{argmin}\{d(q,v) \mid (v,u) \in E\}$, which takes another Δ distance queries. Observe that q must lie in the same part V_i as v, because the shortest between these two vertices does not intersect S.

The next step after the zoom-in process is to find the nearest neighbor itself. Let S' be the union of all the vertices in all the separators encountered during the zoom-in on q. It is easy to verify the following two facts:

- $|S'| \leq O(w \log n)$.
- There exists $u \in S'$ that lies on the shortest-path between q and its nearest neighbor in the dataset D. So q and u have the same nearest neighbor.

Recall that the vertices in S' along with all their nearest neighbors are stored explicitly in the data structure, and they can be "compared" against q using a distance oracle. Hence, it is possible to find q's *exact* nearest neighbor in time $O(\Delta w \log n)$, assuming access to a distance oracle. This proves Theorem 2.

▶ Remark. Both the "zoom-in" phase and the calculation of the nearest neighbor itself were made easy by the fact that we stored all the separators explicitly in the data structure. Planar graphs on m nodes have separators of size $O(\sqrt{m})$, but in our model $m \gg n$, and thus storing the separators explicitly is prohibitively expensive.

2 Planar Graph Metrics

In this section we start proving Theorem 1. Let $G = (M, E)$ be a connected planar graph with positive edge weights $\omega : E \to \mathbb{R}_+$, and let $D \subseteq M$ be the dataset vertices. We denote $n = |D|$, $m = |M|$, and Δ is the maximum degree in G. Assume the minimal edge weight is 1, and let $Diam$ be the diameter of the graph. We use $\mathbf{d_G}(\cdot,\cdot)$ to denote the shortest path distance in G, and assume it can be computed in time t_{DO} e.g., by having access to a distance oracle.

We shall start with the case where shortest paths in G are unique, as it simplifies technical matters considerably. A common workaround to this uniqueness issue is to perturb the edge weights, but this solution is *not applicable* in our model of a black-box access to the distance function $\mathbf{d_G}(\cdot,\cdot)$, because its implementation could potentially exploit ties (e.g., by assuming all distances are small integers). The general case is sketched in Section 3.

It is worth pointing out that our algorithm relies on machinery developed in [1] to recursively partition a planar graph G, which relies in turn on a two-path planar separator. Unlike many algorithms for planar graphs, which use existence of *small* separators, this machinery, described in detail in Section 2.1, uses the fact that the separators are shortest paths, and therefore could be represented succinctly, even if they are large. While this idea had been used before, applying it in the context of NNS (and providing matching lower bounds) requires a considerable amount of technical novelty, as described in Section 2.2.

2.1 Building the Hierarchy tree $\mathcal{T}$

Preprocessing algorithm, step 1. The algorithm fixes some vertex $s \in M$. It then constructs a shortest-path tree T rooted at $s \in M$ by invoking Dijkstra's single-source shortest-path algorithm from s.

Two-path planar separator. We use a version of the well-known Planar Separator Theorem by Lipton and Tarjan [21], where the separator consists of two paths in the shortest-paths tree T of G. In the specific version stated below, we are only required to separate a subset $U \subset M$, and the balance constraint refers to a subset $W \subset U$. As usual, $G[U]$ denotes the subgraph of G induced on $U \subset M(G)$. We apply this theorem recursively using the same shortest-paths tree T. An additional concern for us is that this tree is too large to be stored entirely (it spans all nodes M), hence our algorithm will store partial information that suffices to efficiently perform the Zoom-In and Estimating the Distance operations.

Given a connected planar graph G, we assume throughout it is a plane graph, i.e., accompanied by a specific drawing in the plane, and that it is already triangulated. We let the new edges introduced by the triangulation have infinite weight (hence they do not participate in any shortest-path). While the triangulation operation may increase the maximum degree, it will not affect our runtime bounds (which depend on Δ), because our runtime bounds depend on the maximum degree in the tree T, which contains only edges from the original graph (and not from the triangulated one).[1]

For a rooted spanning tree $\tilde{T}$ of G, define the *root-path* of a vertex $v \in M$ in this tree, denoted $\tilde{T}_v$, to be the path in the tree $\tilde{T}$ connecting v to the root. (We use here $\tilde{T}$ for generality, but will soon instantiate it with the tree T constructed in step 1.) The next theorem has essentially the same proof as of [21, Lemma 2]. It is particularly convenient for a recursive application, where $U \subset M$ is the "current" subset to work on, yet G and the tree $\tilde{T}$ remain fixed through the recursive process. It shows that each set could be partitioned using a cycle which composed of two paths in the shortest path tree, plus an edge called the *separator edge*. We remark that $G[U]$ is not required to be connected.

▶ **Lemma 3** ([21]). *Given a triangulated plane graph $G = (M, E)$, a rooted spanning tree $\tilde{T}$ of G, a subset $U \subset M$, and a vertex subset $W \subseteq U$, one can find in linear time a non-tree edge $(u, v) \in E(G) \setminus E(\tilde{T})$ such that the cycle $\tilde{T}_u \cup \tilde{T}_v \cup \{(u, v)\}$ is a vertex-separator in the following sense: $U \setminus (\tilde{T}_u \cup \tilde{T}_v)$ can be partitioned into two subsets U_1 and U_2 such that (i) each of $U_1 \cap W$ and $U_2 \cap W$ is of size at most $2|W|/3$; and (ii) all paths from a vertex in U_1 to a vertex in U_2 intersect $\tilde{T}_u \cup \tilde{T}_v$ at a vertex.*

Preprocessing algorithm, step 2. We invoke the algorithm of Lemma 3 recursively to construct a "hierarchy" tree $\mathcal{T}$ as follows. Each node μ in $\mathcal{T}$ corresponds to a triple $\langle G(\mu), D(\mu), e(\mu)\rangle$ that records an invokation of Lemma 3 on G:

- $G(\mu)$ records the induced subgraph used as $G[U]$;
- $D(\mu) \subset D$ records the subset of data points used as W;
- the tree T constructed in step 1 is used as $\tilde{T}$ (the same tree for all μ); and
- $e(\mu)$ records the edge of G obtained by this invokation.

The root of $\mathcal{T}$ corresponds to $\langle G, D, e_0\rangle$ where e_0 is the edge obtained by invoking Lemma 3 with $U = M$, $W = D$. Consider now some node μ in $\mathcal{T}$, and let U_1 and U_2 be the two subsets of U obtained from the corresponding invokation of Lemma 3 on $G(\mu)$ and $D(\mu)$. The two children of μ in the tree, denoted μ_i for $i = 1, 2$, correspond, respectively, to the invokations of Lemma 3 on the induced subgraphs $G(\mu_i) = G[U_i]$ with the sets $D(\mu_i) = U_i \cap D$. The recursion stops at a node μ if the corresponding $G(\mu) \cap D = \emptyset$.

Structural properties of the hierarchy tree $\mathcal{T}$. We need several definitions and proofs from [1], which are repeated here for completeness. For a node $\mu \in \mathcal{T}$, let the *level* of

[1] One can also triangulate G while increasing its maximum degree by at most a constant factor [16, Theorem 4.4].

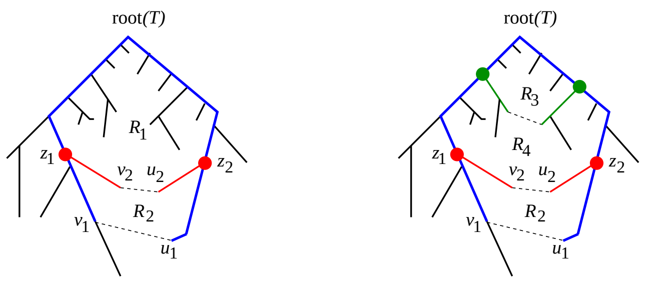

Figure 1 An illustration of apices and frames. The left figure shows the tree T using solid edges. The first separator-edge, at the hierarchy's root, is the dashed edge (v_1, u_1). Let R be the corresponding cycle's interior region. This region R has a separator-edge (v_2, u_2). The cycle-separator of this region R is the union of the two cycles defined by (v_1, u_1) and by (v_2, u_2). The apices of R are labeled z_1 and z_2.
The right figure then shows what happens in region R_1; the separator-edge is a green dashed line, and the new apices are green circles. The frame of R is the subgraph colored blue. The frame of R_2 contains all red and some blue edges. Note that all edges in the first separator $T_{v_1} \cup T_{u_1}$ belong to the frame of only one of R_1 and R_2.

μ, denoted $\textsc{Level}(\mu)$, be the number of edges in $\mathcal{T}$ from μ to the root of $\mathcal{T}$. Clearly, $0 \le \textsc{Level}(\mu) \le 1 + \log_{3/2} n \le 2 \log n$. We now associate with each node μ of $\mathcal{T}$ three subgraphs of G. First, define the *cluster* of μ to be

$$\textsc{Cluster}(\mu) := G(\mu).$$

Second, define the *cycle-separator* of μ recursively as follows. If μ is the root of $\mathcal{T}$, then $\textsc{Cycle-Sep}(\mu) := T_u \cup T_v \cup \{(u, v)\}$. Otherwise, let μ' be the parent of μ in $\mathcal{T}$, and let

$$\textsc{Cycle-Sep}(\mu) := \textsc{Cycle-Sep}(\mu') \cup T_u \cup T_v \cup \{(u, v)\}\,.$$

Third, define the *separator* of μ to be the subgraph of $\textsc{Cycle-Sep}(\mu)$ induced by the edges of T, formally,

$$\textsc{Sep}(\mu) := \textsc{Cycle-Sep}(\mu) \cap E(T).$$

▶ **Observation 4.** *For every μ, the subgraph $\textsc{Sep}(\mu)$ is a subtree of T containing its root s.*

▶ **Observation 5.** *For every μ (other than the root) and its parent μ', the vertices of $\textsc{Sep}(\mu')$ separate $\textsc{Cluster}(\mu)$ from the rest of G. This is immediate from Lemma 3.*

Define the *home* of a vertex $x \in M$, denoted $\textsc{Home}(x)$, as the node μ of $\mathcal{T}$ of smallest level such that x belongs to $\textsc{Sep}(\mu)$.

Define the *apices* of a node μ, denoted $\textsc{Apices}(\mu)$, as the set of vertices in $\textsc{Cycle-Sep}(\mu)$ that have degree ≥ 3; see Figure 1 for an illustration. The apices of μ turn out to be a key enabler of our solution. As we show below, there are very few apices per region and they concisely represent the topological connections between nearby regions (note that degree 2 vertices of $\textsc{Cycle-Sep}(\mu)$ simply form paths between pairs of apices and topologically each such path can be contracted into an edge)

The *new apices* of μ are defined as follows. If μ is the root of $\mathcal{T}$, then $\textsc{NewApices}(\mu) := \textsc{Apices}(\mu)$; otherwise, let $\textsc{NewApices}(\mu) := \textsc{Apices}(\mu) \setminus \textsc{Apices}(\mu')$, where μ' is the parent of μ in $\mathcal{T}$. Intuitively, the new apices of μ are the vertices where the separator of μ "disconnects" from its parent separator μ'.

▶ **Lemma 6** ([1]). *For every μ we have $|\textsc{NewApices}(\mu)| \le 2$.*

We next define the *frame* of a node μ, which is, loosely speaking, a small subgraph of $\textsc{Sep}(\mu')$ that separates $\textsc{Cluster}(\mu)$ from the rest of G. Formally, if μ is the root of $\mathcal{T}$, then $\textsc{Frame}(\mu)$ is the empty graph. Otherwise, let μ' be the parent of μ, and let $\textsc{Frame}(\mu)$ be the subgraph of $\textsc{Sep}(\mu')$ induced by the vertices x in $\textsc{Sep}(\mu')$, which can be connected to (some vertex z in) $\textsc{Cluster}(\mu)$ by a path whose internal vertices (i.e., all but x, z) are all in $\textsc{Sep}(\mu') \setminus \textsc{Apices}(\mu')$. By construction, a path connecting a vertex of $\textsc{Cluster}(\mu)$ to a vertex outside the cluster has to intersect $\textsc{Frame}(\mu)$.

The *region* of μ, denoted $\textsc{Reg}(\mu)$, is the subgraph of G induced by all the vertices of $\textsc{Cluster}(\mu) \cup \textsc{Frame}(\mu)$.

▶ **Lemma 7** ([1]). *For every level $\ell \ge 0$ in $\mathcal{T}$, every edge $e \in E$ belongs to the frame of at most two nodes μ for which $\textsc{Level}(\mu) = \ell$.*

2.2 Finding the Query's Region

Our goal is to find for every level $\ell \le \textsc{Level}(\textsc{Home}(q))$ the region that contains the query q. (There is exactly one such region, because q is not in the separator.) We provide a slightly weaker guarantee that is sufficient for our needs: we show how to compute for every $\ell = 0, 1, \dots, 2\log n$ a set $\mathcal{A}_\ell(q)$ of at most two regions, such that whenever $\ell \le \textsc{Level}(\textsc{Home}(q))$, the set $\mathcal{A}_\ell(q)$ contains the region containing q.

Query Algorithm for Region Finding (Zoom-In). For a node μ of $\mathcal{T}$, let its near-apices be the set of all edges incident to the apices of μ or to s (the root of the shortest path tree T); formally, $NA(\mu) := \{(y, x) \in E \mid x \in \textsc{Apices}(\mu) \cup \{s\}\}$, where we view each (y, x) as an ordered pair.

The query algorithm computes $\mathcal{A}_\ell(q)$ iteratively for level $\ell = 0, 1, \dots, 2\log n$. After initializing $\mathcal{A}_0(q) = \{\text{root}(\mathcal{T})\}$, it computes the next set $\mathcal{A}_\ell(q)$ using $\mathcal{A}_{\ell-1}(q)$ as follows. Find the edge (y, x) for which y is furthest away from s, among all edges $(y, x) \in \cup_{\mu \in \mathcal{A}_{\ell-1}(q)} NA(\mu)$ such that (y, x) is on the shortest path from q to s. Here, we treat (y, x) as an ordered pair, and insist that y appears before x along the path from q to s, or equivalently, that $\mathbf{d_G}(q, y) + \omega(y, x) + \mathbf{d_G}(x, s) = \mathbf{d_G}(q, s)$, which can be checked using only a constant number of distance oracle queries. Notice such (y, x) always exists, because the root s is an apex and one of its incident edges is on the shortest paths from q to s. Next, let $\mathcal{A}_{\ell+1}(q)$ be the set of regions at level $\ell + 1$ that contain the edge (y, x), which we prepare in advance (during the preprocessing phase) as the set $\hat{\mathcal{A}}_{\ell'}((y, x))$. Finally, proceed to the next iteration.

```
algorithm FindRegions ($q$)
0.  let $\mathcal{A}_0(q) = \{\text{root}(\mathcal{T})\}$
1.  for $\ell = 1$ to $2\log n$ do
      a. pick $(y, x) \in E$ of maximal $\mathbf{d_G}(y, s)$ among all $(y, x) \in$
         $\cup_{\mu \in \mathcal{A}_{\ell-1}(q)} NA(\mu)$ that satisfy $\mathbf{d_G}(q, y) + \omega(y, x) + \mathbf{d_G}(x, s) = \mathbf{d_G}(q, s)$
      b. let $\mathcal{A}_\ell(q) = \hat{\mathcal{A}}_\ell((y, x))$
2.  return the sets $\mathcal{A}_\ell(q)$ for $\ell = 0, \dots, 2\log n$
```

Figure 2 Zooming-in on the query's region.

▶ **Lemma 8.** *For every level $\ell \leq \textsc{Level}(\textsc{Home}(q))$, the query vertex q belongs to a region in $\{\textsc{Reg}(\mu) \mid \mu \in \mathcal{A}_\ell(q)\}$.*

Proof. The proof is by induction on ℓ. For $\ell = 0$, we initialized $\mathcal{A}_0(q) = \{\mathrm{root}(\mathcal{T})\}$ and the corresponding region $\textsc{Reg}(\mathrm{root}(\mathcal{T}))$ is the entire graph G, hence $q \in \textsc{Reg}(\mathrm{root}(\mathcal{T}))$. Assume now the claim holds for level ℓ and consider level $\ell + 1$. Let (y, x) be the edge of maximal $\mathbf{d_G}(s, y)$ among all edges in $\cup_{\mu \in \mathcal{A}_\ell(q)} NA(\mu)$ that lie on a shortest path from q to s.

Suppose $q \in R$ for a region R at level $\ell + 1$, and let us show that $R \in \mathcal{A}_\ell(q)$. Let μ be the corresponding node in $\mathcal{T}$, i.e., $\textsc{Reg}(\mu) = R$, and let μ' be the parent node of μ in $\mathcal{T}$. Observe that $q \in \textsc{Reg}(\mu')$, because q must be in $\textsc{Cluster}(\mu)$ (rather than $\textsc{Frame}(\mu)$), and all such vertices are inside the region of the parent μ'. Thus by the induction hypothesis $\mu' \in \mathcal{A}_\ell(q)$.

Let $P(q, s)$ be the unique shortest path from q to s, and let z be the first vertex along this path (furthest from the root s) which is in $\textsc{Apices}(\mu')$. Such z exists (because the root s is an apex) and is not the first vertex on the path (because q itself is not an apex), so let z_1 be vertex preceding z on this path.

We now claim that (z_1, z) is exactly the edge (y, x) chosen. Indeed, the edge (z_1, z) satisfies the two requirements (it is in $NA(\mu')$ and on the shortest path from q to s) by definition, and moreover, z was chosen to that it is closest to q and thus furthest from s.

The claim implies, by the construction of z as the first apex on $P(q, s)$, that the region R containing q also contains the edge $(z_1, z) = (y, x)$, which means that the algorithm will add $R = \textsc{Reg}(\mu)$ to $A_{\ell+1}(q)$. ◀

2.3 Estimating the Distance

Once we have located the region of q, we would like to complete the nearest neighbor search. Let $t^* \in D$ be the closest dataset point to q, and let $P(q, t^*)$ be the shortest path from q to t^*, then we would like to approximate the length of $P(q, t^*)$. Since we located q's region, we can follow the path from the root of the tree $\mathcal{T}$ to q's region. We observe that at some tree node μ we reach a situation where $P(q, t^*)$ intersects $\textsc{Frame}(\mu)$. Indeed, the region at the root of the tree contains both q and t^*; however, at the leaf μ of the tree, the region contains q and either (i) does not contain t^*, in which case the path $P(q, t^*)$ connects a vertex in $\textsc{Cluster}(\mu)$ to one outside $\textsc{Cluster}(\mu)$, and thus must intersect $\textsc{Frame}(\mu)$; or (ii) it does contain t^*, but only in its frame and not in its cluster (because $\textsc{Cluster}(\mu) \cap D = \emptyset$), in which case t^* is itself in the intersection.

If the query procedure could identify a vertex v on this intersection between $P(q, t^*)$ and $\textsc{Frame}(\mu)$, then it could solve finding the nearest neighbor problem for q by finding the nearest neighbor of v and reporting the exact same vertex (and this holds also for approximate nearest neighbor). This is exactly the approach taken in our warmup, the bounded treewidth case, where the preprocessing phase stores for every separator vertex its nearest neighbor in D, and the query procedure just considers all the separator vertices in all the regions encountered during the zoom-in process for q.

However, in the planar graph setting, the number of vertices on a single separator may be arbitrarily large (compared to n). So we must exploit the separator's structure as the union of a few shortest paths. At a very high level, our solution is to carefully choose net-points on the boundary of each region, and only for these net-point we store their nearest neighbor in D. The challenge is to choose the net-points in such a way that (i) for at least one "good" net-point in the sense that the distance from q through the net-point and then to D (i.e., to the nearest neighbor of this net-point) is guaranteed to approximate the optimal NNS answer; and (ii) the query procedure can examine very few net-points (compared to the total number of net-points stored, which is linear in n) until one of these good points is found.

Before describing the algorithm in more detail, let us introduce some useful notations. For a vertex c on a path P and a distance $\rho' > 0$, let $P(c, \rho')$ be all nodes in P at distance at most ρ' from c. Let $\mathcal{N}(P, c, \rho, \rho')$ be a set of nodes in $P(c, \rho')$ such that every node in $P(c, \rho')$ has a node in $\mathcal{N}(P, c, \rho, \rho')$ at distance ρ and every two nodes in $\mathcal{N}(P, c, \rho, \rho')$ are at distance at least ρ from one another (this set can be obtained by considering all nodes on the path $P(c, \rho')$ from endpoint of the path to another and adding to $\mathcal{N}(P, c, \rho, \rho')$ every node that does not have yet a node in $\mathcal{N}(P, c, \rho, \rho')$ at distance ρ from it). For a shortest path P and a node v, let

$$N_i(v, P) := \mathcal{N}(P, c_v, 2^i\epsilon/32, 2^{i+1}),$$

and

$$N(v, P) := \bigcup_{0 \le i \le \log(nDiam)} N_i(v, P),$$

where c_v is the vertex in P closest to v.

For a tree node $\mu \in \mathcal{T}$ such that $e(\mu) = (u, v)$, we let $N(T_u) := \cup_{w \in \textsc{Cluster}(\mu) \cap D} N(w, T_u)$, and similarly for $N(T_v)$. (Formally, it depends also on μ, but we suppress this.) For a shortest path P from s to some vertex f, let $N(P, d_1, d_2)$ to be a vertex in $N(P)$ as follows. If $d_2 > \mathbf{d_G}(s, f)$ then $N(P, d_1, d_2) := f$. If $d_1 < 0$ then $N(P, d_1, d_2) := s$. Otherwise let $N(P, d_1, d_2)$ be the vertex $x \in N(P)$ with minimal $\mathbf{d_G}(x, s)$ among all vertices x satisfying $d_1 \le \mathbf{d_G}(x, s) \le d_2$; if no such vertex x exists, set $N(P, d_1, d_2) := null$. For a tree node μ, let $\mathcal{P}(\mu) = \{T_u\} \cup \{T_v\}$ where $e(\mu) = (u, v)$.

Preprocessing. Let us describe the additional information stored by our data structure. For every node μ, where we denote $(u, v) = e(\mu)$, store the sets $N(T_z)$ for all $z \in \{u, v\}$. In addition, construct a range reporting data structure on $N(T_z)$ according to the distance from s. Namely, a data structure that given two distances d_1, d_2 returns in $O(\log\log n)$ time a vertex $x \in N(T_z)$ with $\mathbf{d_G}(x, s) \in [d_1, d_2]$ that has minimal $\mathbf{d_G}(x, s)$ among all such vertices, or returns null if no such vertex exists. Observe that the range reporting data structure on $N(P)$ makes it possible to find $N(P, d_1, d_2)$ in $O(\log\log n)$ time [4].

In addition, for every node μ, level $\ell \in \{1, \ldots, 2\log n\}$ and apex $x \in \textsc{NewApices}(\mu)$, store for every edge e incident to x the set $\hat{\mathcal{A}}_\ell(e) = \{\mu \in \mathcal{T} \mid e \in E(\textsc{Reg}(\mu)) \textit{ and } \textsc{Level}(\mu) = \ell\}$, namely, the set of level ℓ tree nodes μ for which e belongs to their region (recall there are at most two such nodes). The algorithm also stores the number $O_T(v)$ for every vertex v that is a neighbor of an apex of some node $\mu \in \mathcal{T}$ (for all apices).

For every tree node μ the algorithm stores an indicator $\mathrm{IL}(\mu)$ if μ is a leaf in $\mathcal{T}$. Note that if μ is a leaf in $\mathcal{T}$ then its cluster contains at most one dataset point, denoted by $D(\mu)$. The algorithm also stores the dataset point $D(\mu)$ in case μ is a leaf.

Distance Query. The distance query given a vertex q is performed as follows. (See Figure 3 for a pseudo-code description.) The algorithm starts by invoking Procedure **FindRegions** to obtain the sets $\{\mathcal{A}_i(q)\}$ for $1 \le i \le 2\log n$, where each set $\mathcal{A}_i(q)$ contains at most two nodes of level i in $\mathcal{T}$ such that q belongs to the region of at least one of them. The algorithm then iterates on all path separators P in $\mathcal{P}(q) = \cup_{1 \le i \le 2\log n} \cup_{\mu \in \mathcal{A}_i(q)} \mathcal{P}(\mu)$. For a path $P \in \mathcal{P}(q)$, let $\mu(P)$ be the node such that $P \in \mathcal{P}(\mu)$. For each such path separator P, the algorithm invokes Procedure **DistThroughPath** to estimate $\mathbf{d_G}(q, D \cap \textsc{Cluster}(\mu(P)), P)$, namely, the length of the shortest path from q to some vertex in $D \cap \textsc{Cluster}(\mu(P))$ among all such paths that go through some vertex in P. Let $\tilde{d}(P, q, D)$ be the estimated distance returned by this invokation of Procedure **DistThroughPath**. In addition, the algorithm iterates over

```
algorithm Dist$(q)$
1. $\{\mathcal{A}_i(q)\} \leftarrow$ FindRegions$(q)$.
2. Let $\mathcal{P}(q) = \cup_{\mu \in \mathcal{A}_i(q), 1 \le i \le 2\log n} \mathcal{P}(\mu)$.
3. For every $P \in \mathcal{P}(q)$ do the following:
   a. Set $\tilde{d}(q, D, P) \leftarrow$ DistThroughPath$(q, P)$.
4. Set $d_1 \leftarrow \infty$.
5. For every $\mu \in \cup_{\mu \in \mathcal{A}_i(q), 1 \le i \le 2\log n}$ do:
   a. If IL$(\mu)$ then set $d_1 \leftarrow \min\{d_1, \mathbf{d_G}(q, D(\mu))\}$.
6. Return $\min(\{\tilde{d}(q, D, P) \mid P \in \mathcal{P}(q)\} \cup \{d_1\})$.
```

Figure 3 Our main algorithm for estimating the distance between a given query vertex q and the closest data point to it in D.

```
algorithm DistThroughPath $(q, P)$
1. $found = false$.
2. Set $p \leftarrow s$.
3. While $(found = false)$
   a. Let $\tilde{d} = \mathbf{d_G}(p, q)$.
   b. Find an $\tilde{d}/8$-net $S'$ on $N(P) \cap P(p, 2\tilde{d})$.
   c. Set $p$ to be the vertex in $S' \cup \{p\}$ such that $\mathbf{d_G}(q, p)$ is minimal.
   d. If $\mathbf{d_G}(q, p) > \tilde{d}/2$ then set $found = true$.
4. For $i$ from 1 to $\log nM$ do the following.
   a. Find an $2^i\epsilon/8$-net $S_i$ on $N(P) \cap P(p, 2^{i+3})$.
   b. Set $\tilde{d}(q, D, P)_i$ to be the minimal distance $\mathbf{d_G}(q, x) + \mathbf{d_G}(q, D)$ for $x \in S_i$.
5. Set $\tilde{d}(q, D, P)$ to be the minimal distance $\tilde{d}(q, D, P)_i$.
6. Return $\tilde{d}(q, D, P)$.
```

Figure 4 A procedure for estimating $\mathbf{d_G}(q, D, P)$, which is the minimum length of a path from a given query vertex q to some vertex in D among all such paths that go through some vertex in P.

all nodes $\mu \in \cup_{1 \le i \le 2\log n} \mathcal{A}_i(q)$ to check if μ is a leaf in $\mathcal{T}$, and among all such leaf nodes μ, the algorithm finds the node $\tilde{\mu}$ such that $\mathbf{d_G}(q, D(\tilde{\mu}))$ is minimal, denoting it d_1. (This computation is straightforward, since $|D(\tilde{\mu})| \le 1$ for leaf nodes.) The algorithm then returns $\min(\{d_1\} \cup \{\tilde{d}(q, D, P) \mid P \in \mathcal{P}(q)\})$.

Procedure **DistThroughPath** is given (q, P) and works in two stages. (See Figure 4 for a pseudo-code description.) The first stage finds a vertex $p \in P$ that is "close" to q, and the second one uses this p to compute the estimated distance $\tilde{d}(q, D, P)$. The first stage is done as follows. Initialize $p = s$ and $found = false$, and now while $found = false$, do the following: first, let $\tilde{d} = \mathbf{d_G}(p, q)$; second, find a $\tilde{d}/8$-net S' on $N(P) \cap P(p, 2\tilde{d})$; third, set p to be a vertex in $S' \cup \{p\}$ that minimizes $\mathbf{d_G}(q, p)$; finally, if $\mathbf{d_G}(q, p) > \tilde{d}/2$ then set $found = true$ (namely, the first part is finished).

The second stage is then done as follows. For i from 1 to $\log(nM)$ do the following. First, find a $2^i\epsilon/8$-net S_i on $N(P) \cap P(p, 2^{i+3})$. Second, set $\tilde{d}(q, D, P)_i$ to be the minimal distance $\mathbf{d_G}(q, x) + \mathbf{d_G}(q, D)$ for $x \in S_i$. Now return the minimal distance $\tilde{d}(q, D, P)_i$ as the final answer $\tilde{d}(q, D, P)$.

2.4 Analysis

Recall that $t^* \in D$ is the closest data point to q.

▶ **Lemma 9.** *Consider a node $\mu \in \mathcal{T}$, let $C = \textsc{Cluster}(\mu)$ and $e(\mu) = (u, v)$. Let $P \in \{P_u, P_v\}$. Consider a vertex $x \in P$, there is a vertex $z \in N(P)$ at distance at most $\epsilon\, \mathbf{d_G}(x, D \cap C)/16$ from x.*

Proof. Let $t \in D \cap C$ be the vertex of minimal $\mathbf{d_G}(x, t)$, namely, $\mathbf{d_G}(x, t) = \mathbf{d_G}(x, D \cap C)$. Let i be the index such that $2^{i-1} \leq \mathbf{d_G}(x, t) \leq 2^i$. Note that $\mathbf{d_G}(c_t, x) \leq \mathbf{d_G}(c_t, t) + \mathbf{d_G}(t, x) \leq 2\, \mathbf{d_G}(t, x)$. Hence $x \in P(c_t, 2^{i+1})$.

Recall that $N(P)$ contains $\mathcal{N}(P, c_t, 2^i\epsilon/32, 2^{i+1})$, namely, for every vertex $y \in P(c_t, 2^{i+1})$ there is a vertex $z' \in N(P)$ such that $\mathbf{d_G}(y, z') \leq 2^i\epsilon/32 \leq \epsilon\, \mathbf{d_G}(x, D \cap C)/16$. Hence in particular there is a vertex $z \in N(P)$ at distance at most $\epsilon\, \mathbf{d_G}(x, D \cap C)/16$ from x. ◀

Consider a node $\hat{\mu} \in \mathcal{T}$ such that $t^* \in \textsc{Cluster}(\hat{\mu})$ and let $e(\hat{\mu}) = (\hat{u}, \hat{v})$. Let $P \in \{T_{\hat{u}}, T_{\hat{v}}\}$. Let $\mathbf{d_G}(q, t^*, P)$ be the distance of the shortest path from q to t^* among all q to t^* paths that contain at least one vertex in P. Consider Procedure **DistThroughPath** when invoking on (q, P). Let $p_{final}(P)$ be the vertex p when the algorithm reaches step 4 of Procedure **DistThroughPath** invoked on (q, P).

▶ **Lemma 10.** $\mathbf{d_G}(q, p_{final}(P)) \leq 4\, \mathbf{d_G}(q, t^*, P)$.

Proof. Let c_q be the closest vertex to q in P. Let p_i be the vertex p in the beginning of the i'th iteration of the while loop in step 3 of Procedure **DistThroughPath**.

Note that the algorithm continues to the next iteration as long $\mathbf{d_G}(q, p_{i+1}) \leq \mathbf{d_G}(q, p_i)/2$. Let $p_r = p_{final}(P)$. Note also that $\mathbf{d_G}(q, p_r) > \mathbf{d_G}(q, p_{r-1})/2$. From triangle inequality it follows that $c_q \in P(p_{r-1}, 2\, \mathbf{d_G}(q, p_{r-1}))$.

Let S' be the $\mathbf{d_G}(q, p_{r-1})/8$-net on $N(P) \cap P(p, 2\, \mathbf{d_G}(q, p_{r-1}))$ from step 3*b* of the while loop. If $\mathbf{d_G}(q, p_{r-1}) \leq 4\, \mathbf{d_G}(q, t^*, P)$ then we are done as $\mathbf{d_G}(q, p_r) \leq \mathbf{d_G}(q, p_{r-1})$. Seeking a contradiction assume $\mathbf{d_G}(q, p_{r-1}) > 4\, \mathbf{d_G}(q, t^*, P)$. By Lemma 9, $N(P)$ contains a vertex z_1 at distance $\epsilon\, \mathbf{d_G}(c_q, t^*)/16$ from c_q. Recall that S' is an $\mathbf{d_G}(q, p_{r-1})/8$-net on $N(P) \cap P(p_{r-1}, 2\, \mathbf{d_G}(q, p_{r-1}))$. Hence there is a vertex $z_2 \in S'$ at distance at most $\mathbf{d_G}(q, p_{r-1})/8$ from z_1.

We get that,

$$\begin{aligned}
\mathbf{d_G}(q, p_r) &\leq \mathbf{d_G}(q, z_2) \\
&\leq \mathbf{d_G}(q, c_q) + \mathbf{d_G}(c_q, z_2) \\
&\leq \mathbf{d_G}(q, t^*, P) + \epsilon\, \mathbf{d_G}(c_q, t^*)/16 + \mathbf{d_G}(q, p_{r-1})/8 \\
&\leq \mathbf{d_G}(q, t^*, P) + \epsilon(\mathbf{d_G}(c_q, q) + \mathbf{d_G}(q, t^*))/16 + \mathbf{d_G}(q, p_{r-1})/8 \\
&\leq \mathbf{d_G}(q, t^*, P) + \epsilon(\mathbf{d_G}(q, t^*, P) + \mathbf{d_G}(q, t^*, P))/16 + \mathbf{d_G}(q, p_{r-1})/8 \\
&= \mathbf{d_G}(q, t^*, P)(1 + \epsilon/8) + \mathbf{d_G}(q, p_{r-1})/8 \\
&\leq \mathbf{d_G}(q, p_{r-1})(1 + \epsilon/8)/4 + \mathbf{d_G}(q, p_{r-1})/8 \\
&\leq \mathbf{d_G}(q, p_{r-1})/2,
\end{aligned}$$

contradiction. ◀

Let i be the index such that $2^{i-1} \leq \mathbf{d_G}(q, t^*, P) \leq 2^i$.

▶ **Lemma 11.** *The distance $\tilde{\mathbf{d_G}}(q, D, P)$ returned by the Procedure* **DistThroughPath** *satisfies $\mathbf{d_G}(q, D, P) \leq \tilde{\mathbf{d_G}}(q, D, P) \leq (1 + \epsilon)\, \mathbf{d_G}(q, t^*, P)$.*

Proof. It is not hard to verify that $\mathbf{d_G}(q, D, P) \leq \tilde{\mathbf{d_G}}(q, D, P)$, we therefore only need to show the second direction where $\tilde{\mathbf{d_G}}(q, D, P) \leq (1 + \epsilon)\, \mathbf{d_G}(q, t^*, P)$.

Let $w \in P \cap P(q, t^*, P)$. Note that $\mathbf{d_G}(w, t^*) \leq \mathbf{d_G}(q, t^*, P)$. By Lemma 9 there is a vertex $x \in N(P)$ at distance at most $\epsilon\, \mathbf{d_G}(w, t^*)/16 \leq \epsilon\, \mathbf{d_G}(q, t^*, P)/16$ from w.

We have

$$\begin{aligned} \mathbf{d_G}(x, p_{final}(P)) &\leq \mathbf{d_G}(x, w) + \mathbf{d_G}(w, q) + \mathbf{d_G}(q, p_{final}(P)) \\ &\leq \mathbf{d_G}(x, w) + \mathbf{d_G}(w, q) + 4\, \mathbf{d_G}(q, t^*, P) \\ &\leq \epsilon\, \mathbf{d_G}(q, t^*, P)/16 + \mathbf{d_G}(q, t^*, P) + 4\, \mathbf{d_G}(q, t^*, P) \\ &< 6\, \mathbf{d_G}(q, t^*, P) \\ &\leq 6 \cdot 2^i \\ &< 2^{i+3}, \end{aligned}$$

where the second inequality follows by Lemma 10.

We get that $x \in P(p_{final}(P), 2^{i+3})$. Recall that S_i is an $2^i\epsilon/8$-net on $N(P) \cap P(p_{final}(P), 2^{i+3})$. Hence there is a vertex $x_2 \in S_i$ at distance at most $2^i\epsilon/8$ from x. Note that $\mathbf{d_G}(x_2, w) \leq \mathbf{d_G}(w, x) + \mathbf{d_G}(x, x_2) \leq \epsilon\, \mathbf{d_G}(w, t^*)/16 + 2^i\epsilon/8 \leq \epsilon\, \mathbf{d_G}(q, t^*, P)/2$. We get that

$$\begin{aligned} \tilde{\mathbf{d_G}}(q, D, P) &\leq \mathbf{d_G}(q, x_2) + \mathbf{d_G}(x_2, t^*) \\ &\leq \mathbf{d_G}(q, w) + \mathbf{d_G}(w, x_2) + \mathbf{d_G}(x_2, w) + \mathbf{d_G}(w, t^*) \\ &\leq \mathbf{d_G}(q, t^*, P) + 2\, \mathbf{d_G}(x_2, w) \\ &\leq (1 + \epsilon)\, \mathbf{d_G}(q, t^*, P). \end{aligned}$$

◀

The following lemma shows that the estimated distance returned by the algorithm satisfies the desired stretch.

▶ **Lemma 12.** *The distance $\tilde{\mathbf{d_G}}(q, D)$ returned by the algorithm satisfies $\mathbf{d_G}(q, D) \leq \tilde{\mathbf{d_G}}(q, D) \leq (1 + \epsilon)\, \mathbf{d_G}(q, D)$.*

Proof. it is not hard to verify that $\mathbf{d_G}(q, D) \leq \tilde{\mathbf{d_G}}(q, D)$, we therefore only need to show the other direction, namely, $\tilde{\mathbf{d_G}}(q, D) \leq (1 + \epsilon)\, \mathbf{d_G}(q, D)$. Let μ be the leaf node in $\mathcal{T}$ that contains t^*.

If $q \in \textsc{Reg}(\mu)$, then note that by Lemma 8 $\mu \in \{\mu \in \mathcal{A}_i(q) \mid 1 \leq i \leq 2 \log n\}$, therefore the algorithm examines the distance $\mathbf{d_G}(q, D(\mu))$ and returns it if this is the minimal distance examined by the algorithm. We get that $\tilde{\mathbf{d_G}}(q, D) \leq \mathbf{d_G}(q, D(\mu)) = \mathbf{d_G}(q, D)$. So assume $q \notin \textsc{Reg}(\mu)$. Notice that there must be an ancestor node μ' such that $P(q, t^*) \cap P \neq \emptyset$ for some $P \in \{T_u, T_v\}$ where $e(\mu') = (u, v)$. Notice that $P \in \mathcal{P}(q)$ and thus by the algorithm and Lemma 11 we have $\tilde{\mathbf{d_G}}(q, D) \leq (1 + \epsilon)\, \mathbf{d_G}(q, D, P) = (1 + \epsilon)\, \mathbf{d_G}(q, D)$. ◀

▶ **Lemma 13.** *The query algorithm runs in time $O(\frac{1}{\epsilon} \cdot \log\log n + t_{DO}) \log n \log Diam + \log n \cdot \Delta t_{DO})$.*

Proof. Let us start with bounding the time to find the sets $\mathcal{A}_\ell(q)$ for $1 \leq \ell \leq 2 \log n$ in Procedure **FindRegions**. Recall that in order to find the sets $\mathcal{A}_{\ell+1}(q)$ the algorithm examines all $(y', x') \in \cup_{\mu \in \mathcal{A}_\ell(q)} NA(\mu)$ and check which ones satisfy $\mathbf{d_G}(q, y') + \omega(y', x') + \mathbf{d_G}(x', s) = \mathbf{d_G}(q, s)$, and among the ones that satisfy the equality, the algorithm picks the edge $e = (y, x)$ of minimal $\hat{O}_T(e)$. Checking if an edge (y', x') satisfy $\mathbf{d_G}(q, y') + \omega(y', x') + \mathbf{d_G}(x', s) = \mathbf{d_G}(q, s)$ can be done by constant queries to the distance oracle and thus takes $O(t_{DO})$ time.

Let $\tilde{\mu}$ be a node in $\mathcal{A}_{\ell+1}(q)$ and let $\tilde{\mu}'$ be its parent in $\mathcal{T}$. Recall that $\tilde{\mu}' \in \mathcal{A}_\ell(q)$. Recall also that $NA(\tilde{\mu})$ is the set of all edges incident to the apices of $\tilde{\mu}$ or to s. Since $\tilde{\mu}$ is a child of $\tilde{\mu}'$ we have $\textsc{Apices}(\tilde{\mu}) \subseteq \textsc{Apices}(\tilde{\mu}') \cup \textsc{NewApices}(\tilde{\mu})$.

For level j let $e = (x_j, y_j) \in \cup_{\mu \in \mathcal{A}_j(q)} NA(\mu)$ be the edge of minimal $O_T(y_j)$ among all edges $(x', y') \in \cup_{\mu \in \mathcal{A}_j(q)} NA(\mu)$ that satisfy $\mathbf{d_G}(q, y') + \omega(y', x') + \mathbf{d_G}(x', s) = \mathbf{d_G}(q, s)$.

Let $NNA(\mu)$ be the set of all edges incident to the new apices of μ. It is not hard to verify that in order to find $e = (x_{j+1}, y_{j+1})$ given the edge $e = (x_j, y_j)$, it is enough to find the edge (x, y) of minimal $O_T(y)$ among all edges $(x', y') \in \cup_{\mu \in \mathcal{A}_j(q)} NNA(\mu)$ that satisfy $\mathbf{d_G}(q, y') + \omega(y', x') + \mathbf{d_G}(x', s) = \mathbf{d_G}(q, s)$ and compare it with (x_j, y_j). Recall that $\mathcal{A}_{j+1}(q)$ contains at most two nodes. Hence the time spend for level $j+1$ is $O(\Delta \cdot t_{DO})$. Hence the total time to find all sets $\mathcal{A}_\ell(q)$ is $O(\log n \Delta \cdot t_{DO})$.

We now turn to bound the running time of Procedure **DistThroughPath** invoked on (q, P). Recall that Procedure **DistThroughPath** has two main parts. The first part finds a vertex p_{final} such that $\mathbf{d_G}(q, p_{final}(P)) \leq 4\,\mathbf{d_G}(q, t^*, P)$ and the second part uses $p_{final}(P)$ to find an estimation on $\mathbf{d_G}(q, D, P)$.

Let p_i be the vertex p in the beginning of the i'th iteration of the while loop in step 3 of Procedure **DistThroughPath**. The first part is done in iterations, where the algorithm continues to the next iteration i as long as $\mathbf{d_G}(q, p_i) \leq \mathbf{d_G}(q, p_{i-1})$. Therefore the number of iteration is $O(\log Diam)$. It is not hard to see that the time of iteration i is dominated by the maximum of the time for finding a $\tilde{d}/8$-net S' on $N(P) \cap P(p_i, 2\tilde{d})$ and the time for invoking the distance oracle a constant number of times. Finding a $\tilde{d}/8$-net S' on $N(P) \cap P(p_i, 2\tilde{d})$ can be done by $O(\log \log n)$ using the range reporting data structure on $N(P)$ as follows.

For j from -16 to 15, find $N(P, \mathbf{d_G}(s, p_i) + j\tilde{d}/8, \mathbf{d_G}(s, p_i) + (j+1)\tilde{d}/8)$ and add it to S' (initially set to be empty). It is not hard to verify that S' is indeed $\tilde{d}/8$-net on $N(P) \cap P(p_i, 2\tilde{d})$.

The time for a single invokation of the range reporting data structure takes $O(\log \log n)$. Note that the range reporting data structure is invoked a constant number of times. We get that each iteration of the first part takes $O(\log \log n + t_{DO})$ time.

Hence the first part takes $O((\log \log n + t_{DO}) \log Diam)$ time.

Let us now turn to the second part of Procedure **DistThroughPath**. The second part consists of $\log nM = O(\log Diam)$ iterations. It is not hard to see that the time of each iteration is dominated by the maximum of the time for finding a $2^i \epsilon/8$-net S_i on $N(P) \cap P(p, 2^{i+3})$ and the time for invoking the distance oracle $O(1/\epsilon)$ times.

Similarly as explained in the first part finding a $2^i \epsilon/8$-net S_i on $N(P) \cap P(p, 2^{i+3})$ can be done in $O(1/\epsilon \log \log n)$ time. Thus the total time for the second part is $O((1/\epsilon \log \log n + t_{DO}) \log Diam)$ time. We get that the total time for Procedure **DistThroughPath** is $O((1/\epsilon \log \log n + t_{DO}) \log Diam)$.

Finally, we turn to bound the running time of Procedure **Dist**. Procedure **Dist** starts by invoking Procedure **FindRegions** to obtain the sets $\{\mathcal{A}_i(q)\}$ for $1 \leq i \leq 2 \log n$. This takes $O(\log n \Delta \cdot t_{DO})$ as explained above.

The algorithm then iterates on all path separators P in $\mathcal{P}(q) = \cup_{\mu \in \mathcal{A}_i(q), 1 \leq i \leq 2 \log n} \mathcal{P}(\mu)$. Recall that there are at most $O(\log n)$ such paths. For each such path P the algorithm invokes Procedure **DistThroughPath** which takes $O((1/\epsilon \log \log n + t_{DO}) \log Diam)$ time. In addition, the algorithm iterates over all nodes $\mu \in \cup_{\mu \in \mathcal{A}_i(q), 1 \leq i \leq 2 \log n}$ and invokes the distance oracle a constant number of items for each iteration.

We get that the total running time of Procedure **Dist** is $O((1/\epsilon \log \log n + t_{DO}) \log n \log Diam + \log n \Delta \cdot t_{DO})$. ◀

▶ **Lemma 14.** *The space requirement of the data structure is* $O(\frac{1}{\epsilon} n \log n \log Diam + n \Delta \log^2 n)$.

Proof. The number of nodes in $\mathcal{T}$ is $O(n \log n)$. It is not hard to verify that the depth of $\mathcal{T}$ is $O(\log n)$ as for every node μ with parent node μ' we have $\textsc{Cluster}(\mu) \cap D \leq$

$2/3 \cdot \textsc{Cluster}(\mu') \cap D$. In addition, the number of nodes in each level is at most n as the clusters of the nodes are disjoint and the cluster of each node contains a vertex in D.

For every node μ, every level ℓ and every apex $x \in \textsc{NewApices}(\mu)$, the algorithm stores for every edge e that is incident to x the set of at most two nodes $\hat{\mathcal{A}}_\ell(e) = \{\mu \in \mathcal{T} \mid e \in E(\textsc{Reg}(\mu))$ *and* $\textsc{Level}(\mu) = \ell\}$. The algorithm also stores the number $O_T(v)$ for every vertex v that is a neighbor of an apex of some node $\mu \in \mathcal{T}$.

There are at most two apices in $\textsc{NewApices}(\mu)$. For each such apex x there are most Δ incident edges. For each such edge e and for each level ℓ the size of $\hat{\mathcal{A}}_\ell(e)$ is two. We get that the size stored for each node μ for this part is $O(\Delta \log n)$. There are at most $O(n \log n)$ nodes. Thus the total size for this part is $O(n\Delta \log^2 n)$.

For every node μ such that $e(\mu) = (u, v)$, the algorithm stores the sets $N(T_z)$ in an increasing distance from s for $z \in \{u, v\}$. In addition, construct a range reporting data structure on $N(T_z)$ according to the distance from s. The size of the range reporting data structure is $|N(T_z)|$. We thus need to bound the size of all $N(P)$ for all path separators P.

Every vertex $w \in D$ belongs to the clusters of at most $2 \log n$ nodes μ. For each such node μ such that $e(\mu) = (u, v)$, w contributes at most $O(\log \mathit{Diam}/\epsilon)$ vertices to $N(T_z)$. We get that the sum of the sizes of all $N(P)$ for all path separators P is $O(n \log n \log \mathit{Diam}/\epsilon)$. Thus the total size for this part is $O(n \log n \log \mathit{Diam}/\epsilon)$.

Finally, for every node μ the algorithm stores an indicator $\mathrm{IL}(\mu)$ if μ is a leaf in $\mathcal{T}$. The algorithm also stores the data-point $D(\mu)$ in case μ is a leaf. Thus the total size for this part is $O(n \log n)$. Overall, we get that the total size of the data structure is $O(n \log n \log \mathit{Diam}/\epsilon + n\Delta \log^2 n)$. ◀

3 The General Case: Non-Unique Shortest Paths

In this section we show how to handle the general and seemingly much more involved case of non-unique shortest paths. As mentioned above, the common workaround of perturbing the edge weights is not applicable here because we assume only a black-box access to a distance oracle. The main challenge is to efficiently perform the zoom-in operation. In the unique shortest paths case, if we found a node x that is on the shortest path from q to s, then we knew that x is an ancestor of q in the tree T. This provided us with a better idea on where the query q is and and thus we could zoom in to the right regions. The main idea in handling the non-unique case is to have a consistent way of breaking ties in the preprocessing phase while constructing the shortest path tree T. This also considerably complicates the analysis of the zoom-in operation, and we need to use planarity to show that our consistent way of breaking the ties together with planarity is enough to be able to zoom-in correctly (it is easy to create examples where the graph is not planar and then our way of breaking the ties does not give us more information on where the query q is).

Let us start with the modifications needed in the preprocessing phase. We will later show the modifications needed in the zoom-in operation and in the analysis.

3.1 Preprocessing: The General Case

The main difference in the preprocessing phase is in the way the algorithm chooses the shortest path tree T. The definitions below provide a consistent way of breaking such ties, and will be used later extensively.

Identifiers. Fix some vertex $s \in M$, and assign each vertex $v \in M$ a unique *identifier* $\mathbf{id}(v) \in [1..m]$, such that for all $v_1, v_2 \in M$ with $\mathbf{d_G}(s, v_1) < \mathbf{d_G}(s, v_2)$, we have $\mathbf{id}(v_1) >$

$\mathbf{id}(v_2)$. Such identifiers can be computed easily by ordering the vertices according to their distance from s, breaking ties arbitrarily.

Partial-order on shortest paths. The unique identifiers and $s \in M$ induce the following partial order $\lessdot$ on shortest paths in the graph. Let $P = (s = z_1, z_2, \ldots, z_r)$ and $P' = (s = z'_1, z'_2, \ldots, z'_{r'})$ be two shortest paths originating from the same vertex s. (Our definition below actually extends to every two shortest paths, but we will only need the case $z_1 = z'_1 = s$.) We say that P *is smaller than* P' *with respect to* $\lessdot$, denoted $P \lessdot P'$, if the smallest index $j \geq 1$ for which $z_j \neq z'_j$, satisfies $\mathbf{id}(z_j) < \mathbf{id}(z'_j)$. If no such index j exists, which happens if P is a subpath of P' or the other way around, then the two paths are incomparable under $\lessdot$. A shortest path P from $s \in M$ to $v \in M$ is called *minimal with respect to* $\lessdot$ if it is smaller with respect to $\lessdot$ than every other shortest path from s to v. Observe that for every $v \in M$, every two non-identical shortest paths from s to v are comparable, and thus exactly one of all these shortest paths is minimal. We remark that in the above description, and also in the foregoing discussion, it is convenient to implicitly consider paths as "directed" from one endpoint to the other one (usually going further away from s).

Tree with ordered shortest paths. Let T be a shortest-path tree rooted at the fixed vertex $s \in M$, and let $P(s, v, T)$ denote the path in the tree from s to vertex $v \in M$. We say that the tree T is *minimal with respect to* $\lessdot$ if for every vertex $v \in M$ the path $P(s, v, T)$, which is obviously a shortest path, is minimal with respect to $\lessdot$.

Preprocessing algorithm, step 1'. The algorithm fixes some vertex $s \in M$, and gives the vertices unique identifiers as described above. It then constructs a shortest-path tree T rooted at $s \in M$ that is minimal with respect to $\lessdot$, by invoking Dijkstra's single-source shortest-path algorithm from s, with the following slight modification. When there is a tie, namely, the algorithm has to choose an edge (x, y) that minimizes $\mathbf{d_G}(s, x, T) + \omega(x, y)$, then among all the edges achieving the minimum, the algorithm selects the (unique) one for which the path $P(s, x_i, T)$ is minimal with respect to $\lessdot$.

▶ **Claim 3.1.** *A tree T constructed as above is indeed minimal with respect to $\lessdot$.*

We now define a total order on the vertices induced by $\lessdot$ and T as follows. We say that $v \lessdot_T u$ if either (i) v and u are not related and $P(s, v, T) \lessdot P(s, u, T)$; or (ii) v and u are related and v is a descendant of u. The algorithm assigns every vertex v a number $O_T(v)$ from $[1..m]$ such that $O_T(u) < O_T(v)$ iff $u \lessdot_T v$.

In addition, the algorithm assigns every ordered edge $e = (y, x)$ a number $\hat{O}_T(e) \in [1..3m]$ such that for two edges $e = (y, x)$ and $e' = (y', x')$, we have $\hat{O}_T(e) < \hat{O}_T(e')$ iff $O_T(x) < O_T(x')$ or $O_T(x) = O_T(x')$ and $O_T(y) < O_T(y')$.

The rest of the preprocessing phase is similar to the unique-distances case, with the slight modification that every vertex v (resp., edge e) the algorithm stores, it also stores $O_T(v)$ (resp., $\hat{O}_T(e)$).

3.2 Finding the Query's Region: The General Case

In this section we describe the modifications needed in the zoom-in operation for the general non-unique case. The main difference is in the analysis of the zoom-in operation.

The only modification to the zoom-in operation is as follows. Instead of picking the edge $(y, x) \in \cup_{\mu \in \mathcal{A}_\ell(q)} NA(\mu)$ such that (y, x) is on any shortest path from q to s of maximum

$\mathbf{d_G}(s,x)$ and zooming in to regions of (y,x) on the next level, the algorithm picks the edge (y,x) with minimal $O_T(y)$ among all edges $(y,x) \in \cup_{\mu\in\mathcal{A}_\ell(q)} NA(\mu)$ such that (y,x) is on any shortest path from q to s (not necessarily the path in T).

The following main lemma proves the correctness of the zoom-in operation for the non-unique case (the proof is quite technical and is omitted from this version).

▶ **Lemma 15.** *For every level $\ell \le \textsc{Level}(\textsc{Home}(q))$, the query vertex q belongs to a region in $\{\textsc{Reg}(\mu) \mid \mu \in \mathcal{A}_\ell(q)\}$.*

4 Lower Bounds

Our approximate NNS scheme, presented in Theorem 1, requires access to an *exact* (rather than approximate) distance oracle, and its space and time complexity bounds depend linearly on the graph's maximum degree Δ. In this section we prove that these two requirements are necessary. The graphs used in our lower bounds are in fact trees (and thus certainly planar). Let $\mathbf{DO}(u,v)$ denote (the answer for) a distance-oracle probe for the distance between points u, v.

4.1 Linear Dependence on the Degree Δ

We first assume access to an exact distance oracle, and prove a lower bound on the NNS worst-case query time, assuming that the space requirement is not prohibitively large. We actually prove a stronger assertion, and bound the NNS query time only by the number of distance-oracle probes, regardless of any other operations; in particular, we allow the NNS query procedure to read the entire data structure!

Consider a c-approximate NNS (randomized) scheme with the following guarantee: When given a planar graph with N vertices and maximum degree $\log N \le \Delta \le n$, together with a dataset of n vertices, it produces a data structure of size s. Using this data structure, for every query vertex q, with probability at least $1/2$ it finds q's $c-$approximate nearest neighbor using at most t distance-oracle probes. We are interested in the setting where $N \gg n$, say $N \ge n^2$. The following theorem shows that unless s is huge, the query time t must grow linearly with the maximum degree Δ. Let us justify the above requirements on Δ; the assumption $\Delta \le n$ is necessary because $t \le n$ is always achievable, by answering NNS queries using exhaustive search (with no preprocessing); the assumption $\Delta \ge \log N$ is for ease of exposition, and can probably be removed with some extra technical work.

▶ **Theorem 16.** *If $s \le O(N/(\Delta \log_\Delta n))$ bits, then $t \ge \Omega(\Delta \log_\Delta n)$.*

Outline. We prove the theorem by presenting a single distribution over inputs, which is "hard" for all *deterministic* algorithms. That is, every deterministic algorithm is unlikely to succeed in producing a correct answer, under certain space/time constraints (Lemma 20). A bound for randomized algorithms is achieved by fixing the best possible coins (the easy direction of Yao's minimax principle).

The bound for deterministic algorithms is obtained in three steps. First we assume there is no data structure, i.e., memory size $s = 0$, and show that no deterministic algorithm can succeed with more than a constant probability (Lemma 18). We then amplify the bound by considering a series of query points (Lemma 19), at which point the success probability is so tiny that a small data structure cannot help.

4.1.1 The hard distribution

We specify a distribution over NNS instances, namely, a distribution over tree graphs T of size roughly N and degree roughly Δ, data sets D of size n and a query points q. All but the last level of the tree would be fixed, and the randomization occurs only in the way the leaves are connected.

The fixed part of T: Start with a complete tree of arity Δ and exactly N leaves, which means the tree's depth is $H := \log_\Delta N$. (We shall assume for simplicity that all values are integral, to avoid the standard yet tedious rounding issues.) The dataset D is formed by the n vertices at depth (also called level) $h := \log_\Delta n$, and they are labeled $p_1, \ldots, p_n$. Let all edges have unit length, except for the edges at level h, which have length cH. (To extend our results to unweighted trees, replace these edges with paths of corresponding length.) In particular, the distance between every two distinct dataset points is at least $2cH$. Since this part of the tree is fixed, we assume the algorithm "knows it", i.e., it can compute distances without any distance-oracle probes.

The random part of T: The last level $H + 1$ of the tree is random; it is constructed by hanging N leaves labeled $\ell_1, \ldots, \ell_N$ independently at random. In other words, for each vertex ℓ_i we sample uniformly at random one of the N/Δ nodes at level $H - 1$ and connect to it. By standard tail bounds, with probability greater than $1 - 1/n$, at most 2Δ leaves are attached to the same node, so the maximum degree of the graph is $\leq 2\Delta$. We note that this is the only place where we use that $\Delta \geq \log N$. We denote this input distribution by $\mathcal{T}$.

Finally, we need to specify the distribution of query points. Throughout our analysis the query point is chosen uniformly at random from the leaves, namely, a vertex ℓ_q for uniformly random $q \in [N]$. Observe that the nearest neighbor of ℓ_q is the dataset point p_i which is the unique ancestor of ℓ_q at level h. In fact, this p_i is the unique c-approximate nearest neighbor, because $d(\ell_q, p_i) = H - h$ while for $i' \neq i$ we have $d(\ell_q, p_{i'}) \geq 2cH$. Thus, in all these instances, exact NNS is equivalent to c-approximate NNS.

4.1.2 The no-preprocessing case

Let A be a deterministic algorithm that solves c-approximate NNS without any preprocessing, in other words, A has zero space requirements and consists of only a query algorithm. Define T_A as the number of distance-oracle probes that A makes given a query. Under the above input distribution, T_A is a random variable, and our goal is to show that it is likely to be $\Omega(\Delta \log_\Delta n)$. Towards this end, we shall make a few adaptations to T_A and to the algorithm A.

Let T'_A be the number of distance-oracle probes of the form $\mathbf{DO}(\ell_q, \cdot)$. Clearly $T'_A \leq T_A$ so it suffices to bound T'_A. We next show that in effect, we may restrict attention to algorithms that do not probe the distance from ℓ_q to vertices at level bigger than h (i.e., strict descendants of the dataset D).

▶ **Lemma 17.** *There is an algorithm A_1 that probes $\mathbf{DO}(\ell_q, w)$ only for vertices w at level at most h, and with probability 1 (i.e., on every instance in the support), $T'_{A_1} \leq T'_A$.*

Proof. Algorithm A_1 simulates A probe by probe, except that when A probes $\mathbf{DO}(\ell_q, w)$ for some w at level bigger than h, algorithm A_1 probes $\mathbf{DO}(\ell_q, p_i)$ where p_i is the dataset point which is the ancestor of w. Now, if p_i is also the ancestor of ℓ_q then p_i is the nearest neighbor and A_1 can output p_i. Otherwise observe that $d(\ell_q, w) = d(\ell_q, p_i) + d(p_i, w)$, so A_1

can compute $d(\ell_q, w)$ and continue the simulation of A. Since A_1 uses one query to simulate a query of A, clearly $T'_{A_1} \leq T'_A$.

The remaining thing to specify is how does A_1 find the ancestor of w. Now, if w is not a leaf, then it is part of the fixed graph, and the ancestor is hard-wired into A_1. If w is a leaf, we will assume that the parent p_i is just given to A_1 for free. Formally we allow A_1 to query $DO(w, p_j)$ for various j's until w's ancestor is found, and note that these queries are not counted in T'_{A_1}. ◀

For illustration, consider the case where $n = \Delta$ and $N = n^2$, which means that the tree has depth $H = 2$, and the dataset D lies at level $h = 1$. The algorithm A_1 is given a label for a leaf ℓ_q and has to find its parent. We only consider distance-oracle probes of the form $\mathbf{DO}(\ell_q, p_i)$, which have value 1 if p_i is the nearest neighbor of ℓ_q, and value 3 otherwise. Observe that queries that don't involve ℓ_q carry no information on the parent of ℓ_q, and in queries of the form $\mathbf{DO}(\ell_q, \ell_{q'})$, we effectively replace $\ell_{q'}$ with its parent. The situation is thus identical to searching in an unsorted array of size n. The algorithm "scans" the vertices p_i in some order, which is not deterministic (as it might depend on parents of other leaves) but is independent of the correct answer (the parent of ℓ_q). Therefore, T'_{A_1} is distributed uniformly in $[\Delta]$.

We now return to the general hard distribution $\mathcal{T}$ which follows the same intuition but requires additional technical maneuvers. Our goal is to show that without loss of generality, A_1 could be thought as finding the ancestors of q level by level, starting from the root at level 0 and proceeding down to level $h = \log_\Delta n$. Each level requires a search over Δ items, hence we will obtain a lower bound of $\Omega(\Delta h)$.

Given algorithm A_1, define a new algorithm A_2 as follows. Simulate A_1, but whenever A_1 probes $\mathbf{DO}(\ell_q, w)$, probe instead $\mathbf{DO}(\ell_q, w')$, where w' is the minimum-level ancestor of w for which $\mathbf{DO}(\ell_q, w')$ wasn't probed yet. Now, based on the answer, detect whether ℓ_q is a descendant of w' and proceed according to the case at hand:

- If ℓ_q is a not descendant of w', proceed in the simulation of A_1 by calculating the distance $d(\ell_q, w) = d(\ell_q, w') + d(w', w)$ without probing $\mathbf{DO}(\ell_q, w)$ directly.
- If ℓ_q is a descendant of w', probe the entire path from w' to w (namely, the distance between ℓ_q and each vertex along this path) until you can compute $d(\ell_q, w)$, which could happen by reaching either w itself or a vertex which is not an ancestor of ℓ_q.

We point out two crucial observations. First, A_2 recovers the ancestors of ℓ_q one by one starting from level 0 (the root) down to level h (some dataset point). Second, the extra probes are along the path from the root to ℓ_q, with at most one probe outside that path at each level. This is done only up to level h and without repeating the same probe. Thus in total, A_2 always makes at most $2h = 2\log_\Delta n$ more probes of the form $\mathbf{DO}(\ell_q, \cdot)$ than A_1 does, i.e., $T'_{A_2} \leq T'_{A_1} + 2h$. The next lemma analyzes this "well-behaved" algorithm A_2.

▶ **Lemma 18.** $\Pr_{\mathcal{T},q}[T'_{A_2} \leq h\Delta/3] \leq 1/16$.

Proof. Let $w_0, w_1, \ldots, w_h$ denote the ancestors of ℓ_q from level 0 to level h (e.g., w_0 is the root and $w_h \in D$). We say A_2 *recovers* w_i, the first time it queries $DO(\ell_q, w_i)$ and we recall that a key feature of A_2 is that it recovers these vertices one by one.

Denote by X_j, for $j \in [h]$, the number of distance-oracle probes of the form $\mathbf{DO}(\ell_q, \cdot)$ that A_2 makes after recovering w_{j-1} and until recovering w_j. The main observation is that X_j dominates a random value chosen uniformly from $[\Delta]$, even when conditioned on the sequence of probes made prior to recovering w_{j-1}. Indeed, when sampling the location of ℓ_q, the decision which child of w_{j-1} is the ancestor of ℓ_q could be deferred to the moment the children of w_{j-1} are being probed. Hence algorithm A_2 is essentially performing an exhaustive search, akin to searching in an unsorted array. It follows that $\mathbb{E}[X_j] \geq \Delta/2$ and $\mathbb{E}[T'_{A_2}] \geq h\Delta/2$. Moreover, by applying Azuma's inequality (assuming n is large enough), $\Pr[T'_{A_2} \geq h\Delta/3] \leq 1/16$. ◀

Recalling that $T_A \geq T'_A \geq T'_{A_1} \geq T'_{A_2} - 2h$ (always) and assuming $\Delta \geq 24$, we obtain using Lemma 18 that

$$\Pr_{\mathcal{T},q}\left[T_A \leq h\Delta/4\right] \leq \Pr_{\mathcal{T},q}\left[T'_{A_2} - 2h \leq h\Delta/4\right] \leq 1/16. \tag{1}$$

A sequence of queries (with no preprocessing). We say algorithm A (with no preprocessing) *succeeds* on a query ℓ_q if it outputs a correct answer and makes at most $h\Delta/4$ distance-oracle probes. Eq. (1) states that $\Pr_{\mathcal{T},q}[A \text{ succeeds on query } \ell_q] \leq 1/16$. In order to extend the argument to the case with preprocessing we need to decrease that probability to be exponentially small, which we achieve by looking at a sequence of several queries. Let $q_1, \ldots, q_m \in [N]$ be chosen uniformly at random and independently.

▶ **Lemma 19.** *For every $m \leq N/(4h\Delta)$,*

$$\Pr_{\mathcal{T},q_1,\ldots,q_m}[A \text{ succeeds on all queries } \ell_{q_1}, \ldots, \ell_{q_m}] \leq (\tfrac{1}{8})^m.$$

Proof. The main difficulty here is that there might be dependencies between different query points, e.g., if the algorithm's first probe is $\mathbf{DO}(\ell_6, \ell_7)$, then there is a chance that $q_1 = 6$ and $q_2 = 7$, and we cannot argue the success of A on q_1 and on q_2 are independent. In particular, we cannot assume that A never probes other leaves (other than the query point).

The way we handle it is by sampling the tree using deferred decisions, meaning that we attach every leaf of the tree only when it is needed. Trace the executions of A on query q_i for $i = 1, \ldots, m$ one by one, where each execution is restricted to at most $h\Delta/4$ distance-oracle probes. Every time a leaf is probed for the first time (more precisely, the distance from/to that leaf), determine its location by attaching it to a random vertex at level $H - 1$.

Now, assume algorithm A succeeded on queries $q_1, \ldots, q_{i-1}$ and consider its execution on q_i. The number of leaves attached prior to this execution is at most $(i-1)h\Delta/4 \leq mh\Delta/4 \leq N/16$. Thus, the probability that a random q_i is one of these leaves is at most $1/16$; if this is not the case, then q_i is completely random leaf, and Eq. (1) applies to it. Thus, assuming A already succeeded on queries $q_1, \ldots, q_{i-1}$ the probability it succeeds on query q_i is at most $1/8$. The theorem follows. ◀

4.1.3 Algorithm with preprocessing

We turn to the case where an algorithm can prepare a data structure of size s, and prove a lower bound under the same input distribution as before. Specifically, an input tree is first drawn from the distribution $\mathcal{T}$, and then the tree is processed to create a data structure of s bits. Next, a random leaf ℓ_q is chosen as a query point, and algorithm B, which can read the entire data structure (as "advice"), answers the query. As before, we say that algorithm B *succeeds* on a query ℓ_q if it outputs a correct answer and makes at most $h\Delta/4$ distance-oracle probes.

▶ **Lemma 20.** *If $s \leq N/(4h\Delta)$ then $\Pr_{\mathcal{T},q}[B \text{ succeeds on a query } q] < 1/2$.*

Proof. Assume for contradiction that B succeeds with probability at least $1/2$. Define a new algorithm A that guesses the s bits of advice at random and then simulates algorithm

B. We will show that this algorithm A (which obviously uses no preprocessing) contradicts Lemma 19.

Now consider sampling a tree from $\mathcal{T}$. In expectation, algorithm B would succeed on at least half the leaves of this tree (as query points), hence by Markov's inequality, with probability at least 1/4 (over the choice of the tree) algorithm B succeeds on at least $1 - \frac{4}{3} \cdot \frac{1}{2} = \frac{1}{3}$ of the leaves. In such a tree, the probability that B succeeds on m random leaves is at least $(\frac{1}{3})^m$.

Recall that Algorithm A guesses the advice strings independently at random. Thus, the probability that A succeeds on all the m leaves of a random tree is at least $(\frac{1}{2})^s \cdot \frac{1}{4} \cdot (\frac{1}{3})^m$. Taking $m = N/(4h\Delta)$ and observing $s \leq m$, we have

$$\Pr_{\mathcal{T}, q_1, \ldots, q_m}[A \text{ succeeds on all queries } \ell_{q_1}, \ldots, \ell_{q_m}] \geq \tfrac{1}{4} \cdot (\tfrac{1}{6})^m.$$

Now observe that for N (and hence m) large enough, we have contradicted Lemma 19. ◀

We can now complete the proof of Theorem 16.

Proof of Theorem 16. Assume towards contradiction there is a randomized algorithm C for the c-approximate NNS problem, such that on tree instances with maximum degree at most $2\Delta + 2$, we have (the theorem would then follow by substituting $\Delta' = 2\Delta + 2$): (a) for each query point, with probability at least 3/4, the algorithm answers correctly (a c-approximate nearest neighbor); (b) the space requirement is $s \leq N/(4h\Delta) = N/(4\Delta \log_\Delta n)$; and (c) for each query point, the algorithm makes at most $t \leq h\Delta/4$ distance-oracle probes.

By fixing the coins of algorithm C optimally, it immediately follows there exists some *deterministic* algorithm B that achieves $\Pr_{\mathcal{T},q}[B \text{ succeeds on a query } q] \geq \frac{3}{4}$, in addition to satisfying the space requirement (b) and query time bound (c), which contradicts Lemma 20. The theorem follows. ◀

4.2 Approximate Distance Oracles

We now sketch the argument claiming it is essential to have an exact distance oracle (rather than an approximate one). Suppose the distance oracle provides a $(1 + \delta)$ multiplicative approximation of the distance for some $\delta > 0$. We show an instance with the following properties:

- The total number of nodes is $O(n^2)$ and the maximum degree is $O(\log n)$.
- The aspect ratio across edge weights is $\max\{(2 \log n)/\delta, (1 + \epsilon) \log n\}$.
- If the space of the data structure is $\leq n^2$ bits then the number of distance-oracle probes needed is $\Omega(n)$.

The construction is as follows. Build a binary tree of height $\log n$ from root s, so the binary tree has n leaves, and call this part of the graph the *top tree*. Now, from each leaf of the top tree hang an edge of length $\max\{2 \log n/\delta, (1 + \epsilon) \log n\}$. To simplify the exposition, assume this maximum is $2 \log n/\delta$. The bottom nodes of these edges are labeled $p_1, \ldots, p_n$ and these are the dataset points. Finally, from each dataset point hang a binary tree of depth $\log n$, so we have a total of n^2 leaves to which we hang random n^2 nodes labeled $l_1, \ldots, l_{n^2}$, thus creating nodes of maximum degree $O(\log n)$ with high probability. We call these trees the *bottom trees*.

Observe that for each leaf l_i there is one p_j for which $d(l_i, p_j) = \log n$ while for $k \neq j$ $d(l_i, p_k) \geq (1 + \epsilon) \log n$. Thus, for a query l_i the algorithm must output p_j, which is the unique $(1+\epsilon)$-approximate nearest neighbor. When probed for $\mathbf{DO}(u, v)$, the distance oracle answer is as follows:

- If both u, v belong to the top tree or to the *same* bottom tree, the oracle answers with the exact distance between them.
- If u belongs to a bottom tree and v belongs to a top tree, the oracle answers $2 \log n/\delta$. Note that the exact distance is in the range $[2 \log n/\delta, (1+\delta)2 \log n/\delta]$.
- If u and v belong to different bottom trees, the oracle outputs $4 \log n/\delta$. Again, observe that the correct distance is in the range $[4 \log n/\delta, (1+\delta)4 \log n/\delta]$.

The way the distance oracle is set up, the NNS algorithm faces a situation which is similar to the case where the root has degree n, and is connected to the all the dataset points by distinct edges. In this case the total size of the graph is n^2. The proof of the previous section essentially shows that unless the data structure is of size roughly n^2, the query time is $\Omega(\Delta) = \Omega(n)$.

References

1 Ittai Abraham, Shiri Chechik, and Cyril Gavoille. Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels. In *Proceedings of the 44th symposium on Theory of Computing*, pages 1199–1218. ACM, 2012.

2 Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical hub labelings for shortest paths. In *Proceedings of the 20th Annual European conference on Algorithms*, ESA'12, pages 24–35. Springer-Verlag, 2012.

3 Ittai Abraham and Cyril Gavoille. Object location using path separators. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC'06, pages 188–197. ACM, 2006.

4 Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *41st Annual Symposium on Foundations of Computer Science*, pages 198–207, 2000.

5 Eyal Amir. Approximation algorithms for treewidth. *Algorithmica*, 56(4):448–479, January 2010.

6 Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

7 Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

8 A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *23rd international conference on Machine learning*, pages 97–104. ACM, 2006.

9 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, December 1996.

10 Hans L. Bodlaender, Pal Gronas Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. An $O(c^k n)$ 5-approximation algorithm for treewidth. In *54th Annual Symposium on Foundations of Computer Science*, pages 499–508, 2013.

11 Hans L. Bodlaender, John R. Gilbert, Hjálmtýr Hafsteinsson, and Ton Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *J. Algorithms*, 18(2):238–255, March 1995.

12 R. Cole and L.-A. Gottlieb. Searching dynamic point sets in spaces with bounded doubling dimension. In *38th Annual ACM Symposium on Theory of Computing*, pages 574–583. ACM, 2006.

13 David Eisenstat, Philip N. Klein, and Claire Mathieu. Approximating k-center in planar graphs. In *25th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 617–627. SIAM, 2014.

14 Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Conference on Experimental Algorithms*, WEA'08, pages 319–333. Springer-Verlag, 2008.

15 S. Har-Peled and M. Mendel. Fast construction of nets in low-dimensional metrics and their applications. *SIAM Journal on Computing*, 35(5):1148–1184, 2006.

16 Goos Kant and Hans L. Bodlaender. Triangulating planar graphs while minimizing the maximum degree. *Inf. Comput.*, 135(1):1–14, May 1997.

17 D. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *34th Annual ACM Symposium on the Theory of Computing*, pages 63–66, 2002.

18 R. Krauthgamer and J. R. Lee. Navigating nets: Simple algorithms for proximity search. In *15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 791–801, January 2004.

19 R. Krauthgamer and J. R. Lee. The black-box complexity of nearest-neighbor search. *Theoret. Comput. Sci.*, 348(2-3):262–276, 2005.

20 R. Krauthgamer, H. Nguyen, and T. Zondiner. Preserving terminal distances using minors. *SIAM Journal on Discrete Mathematics*, 28(1):127–141, 2014.

21 R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36(2):177–189, 1979.

22 C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory Comput. Syst.*, 32(3):241–280, 1999.

23 Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024, November 2004.