COL-Trees: Efficient Hierarchical Object Search in Road Networks

Tenindra Abeywickrama*

Center for Computational Science, RIKEN, Japan

Muhammad Aamir Cheema

Faculty of Information Technology, Monash University, Australia

Sabine Storandt

Department of Computer and Information Science, University of Konstanz, Germany

Abstract

Location-based services rely heavily on efficient methods that search for relevant points-of-interest (POIs) near a given location. A k Nearest Neighbor (kNN) query is one such example that finds the k closest POIs from an agent's location. While most existing techniques focus on retrieving nearby POIs for a single agent, these search heuristics do not translate to many other applications. For example, Aggregate k Nearest Neighbor (AkNN) queries require POIs that are close to *multiple* agents. k Farthest Neighbor (kFN) queries require POIs that are the antithesis of nearest. Such problems naturally benefit from a hierarchical approach, but existing methods rely on Euclidean-based heuristics, which have diminished effectiveness in graphs such as road networks. We propose a novel data structure, COL-Tree (Compacted Object-Landmark Tree), to address this gap by enabling efficient hierarchical graph traversal using a more accurate landmark-based heuristic. We then present query algorithms that utilize COL-Trees to efficiently answer AkNN, kFN, and other queries. In our experiments on real-world and synthetic datasets, we demonstrate that our techniques significantly outperform existing approaches, achieving up to 4 orders of magnitude improvement. Moreover, this comes at a small pre-processing overhead in both theory and practice.

Keywords: road networks, point-of-interest queries, heuristic search

1. Introduction

Finding nearby relevant objects, and in particular points-of-interest (POIs), efficiently is a critical task in a variety of heuristic search and planning applications including, e.g., real-time path planning in video game maps [1] or efficiently retrieving POIs in location-based services. Nearby relevant POIs are typically obtained by using a range query or a k Nearest Neighbor (kNN) query. A range query [2] returns all POIs within a given distance from an agent's location, e.g., find all restaurants within 1km from a user. A kNN query [3] returns the k closest POIs from an agent's location, e.g., find the three nearest gas stations to a taxi driver. The distance metric may be Euclidean distance (i.e., "as the crow flies") or road network distance (i.e., the length of the shortest path). In this paper, we consider road network distance as it is a more accurate and versatile measure of proximity in many real-world applications such as map-based services and is capable of representing a variety of metrics such as travel time, physical distance, toll cost, emissions, etc.

In the context of graphs like road networks, computing shortest path distances (or network distances) is significantly more expensive than Euclidean distance. As a result, efficient heuristics are needed to retrieve

---

candidate POIs to answer the aforementioned queries and these typically rely on two implicit assumptions to reduce the overall cost: (a) the search occurs from a single agent; and (b) the final result POIs are in the general vicinity of the single agent's location. However, there are many important queries where one or both of these assumptions do not hold. For example, an Aggregate k Nearest Neighbor (AkNN) query [4] is a natural extension of a kNN query for *multiple agents*, where POIs are retrieved considering their aggregate distance from all agents. Given a POI p and a set of agents Q, the aggregate distance of p from the agents is $d_{agg}(Q, p) = agg(d(q_i, p), \forall q_i \in Q)$ where $d(q_i, p)$ denotes the road network distance from an agent q_i to p and $agg()$ is an aggregate function. For example, when the aggregate function is sum, $d_{agg}(Q, p) = \sum_{q_i \in Q} d(q_i, p)$ and when the aggregate function is max, $d_{agg}(Q, p) = \max_{q_i \in Q} d(q_i, p)$. An A$k$NN query returns the k POIs with smallest aggregate distances. Another pertinent example is the k Farthest Neighbors (kFN) problem [5]. The opposite of the kNN problem, kFN queries involve retrieving the k farthest POIs from an agent's location. In both AkNN and kFN queries, due to their definition, the result objects are often not among the closest to the agent locations and heuristics that retrieve candidates incrementally from the agents location (as in most existing work), are unlikely to be efficient.

Answering queries that consider multiple agents are highly relevant for many real-world applications. AkNN queries are widely used in transportation, e.g., ride-sharing [6, 7]. For example, a group of friends planning to meet may want to choose a restaurant such that the total distance they need to travel is minimized. They can issue an AkNN query to find k restaurants with the smallest aggregate distances where the aggregate function is sum. Similarly, consider an urban planning problem that requires building a fire station at one of many candidate sites with the goal to minimize the maximum distance of the station from residential areas in the city. In this case, an AkNN ($k = 1$) query can be used to choose a candidate site p such that its aggregate distance (with function max) from the set of residential areas is minimum among all candidate sites. AkNN search also has another important real-world application in *shortest detour queries*, which is provided in many popular navigation apps. Given a source s and a target t, a shortest detour query returns a POI p such that $d(s, p) + d(p, t)$ is minimized. Consider a user who wants to stop at a gas station on their route from home to work. In this case, a shortest detour query helps find the gas station that minimizes the total distance the user needs to travel. Note that this is a special case of an AkNN query where s and t are considered to be the agent locations and the aggregate function is sum.

kFN queries also find several important real-world applications. They can be used to find spatial outliers, for example, to allow a healthcare facility to identify elderly residents that are farthest by travel time for potential relocation or a delivery company may be interested in finding the farthest customers from their warehouse to understand the worst-case delivery times. kFN queries may also be used in *obnoxious facility location* [8] where the goal is to place facilities with less desirable characteristics as far away as possible from a sensitive location (or group of locations [9]). For example, among a set of candidate sites, we may wish to place a potentially dangerous facility as far away by travel time from the center of a city as possible, such as a biological research facility or quarantine station to discourage civilians visiting or a maximum security prison to give authorities additional time to respond before civilians are exposed to any risk from escapees. The POI need not be obnoxious per se, e.g., placing a backup site or vault as far away from an existing site to maximize security. Each of these could be answered by a kFN query and would benefit from increased accuracy based on network distance metrics. Note that the objective is different to AkNN queries with max aggregate function, which is a minmax problem. Farthest neighbor computation is also frequently used as a subroutine in solutions to many other problems, such as hierarchical clustering [10], feed-link queries [11], traveling salesmen or vehicle routing problems [12], and several metric k-center algorithms [13, 14, 15].

Clearly, both AkNN and kFN queries have important and commonly used real-world applications where road network graph-based metrics would help improve solution accuracy. This necessitates efficient methods and, by extension, heuristics to solve them. While we focus on road networks, it is worth noting that these queries have applications in a wide variety of domains and can be used for other types of graphs such as social networks and wireless sensor networks, etc. In the rest of the paper, we use the term query location to refer to the location of an agent whenever clear by context. For ease of exposition, we assume the road network to be undirected (i.e., $d(x, y) = d(y, x)$ for any two points x and y). The proposed techniques can be easily extended for directed road networks (e.g., with one-way roads).

1.1. Limitations of Existing Techniques

The key to efficiently retrieving POIs in road networks is in developing heuristics to retrieve candidate POIs that are most promising. Heuristics to answer AkNN queries have largely been borrowed from kNN search [4, 16, 17], which is problematic in several ways. First, intuitions to find kNNs do not translate to AkNNs. For example, the most efficient kNN heuristic [18] uses a recurrence rule stating that the k-th nearest POI must be adjacent to the $k-1$ nearest POIs. The rule is exploited by storing the nearest POI of every potential query location and adjacency relationships between regions associated with POIs. However, AkNN queries involve retrieving POIs by an aggregate distance from multiple query locations. Thus, the recurrence rule is no longer true and cannot be applied, rendering the heuristic unsuitable. This approach is even less desirable for kFN queries as it would involve evaluating all POIs to determine the farthest.

A better approach to find AkNNs and kFNs is to conduct a hierarchical search on the road network. One AkNN method [4] searches an R-tree [19] containing the POIs in a branch-and-bound manner according to a Euclidean distance heuristic, similar to another kNN technique [2]. R-trees are constructed in a way that recursively divides POIs into subsets by Minimum Bounding Rectangles (MBRs). During search, a lower-bound aggregate distance for all POIs in a child R-tree node is computed using the Euclidean distances to its MBR. Now the most promising tree branches can be visited to pinpoint result POIs. However, this is not ideal for road networks as Euclidean distance is only a loose lower-bound especially on metrics like travel time, making the heuristic less efficient. Moreover, the inefficiency is exacerbated for AkNNs as the error will also be aggregated, e.g., making it harder to distinguish between candidates. Furthermore, this method is not admissible for kFNs, and cannot be used to answer such queries at all. Although a maximum Euclidean distance could be computed to MBRs, Euclidean distance is still a lower-bound. kFN being a maximization problem, candidates must be retrieved by upper-bound to terminate optimally.

Landmark Lower-Bounds (LLBs) are a more accurate alternative to Euclidean distance [20]. They involve pre-computing and storing certain distances to *landmarks* in the road network, which can then be used to compute a lower-bound between any two locations using the triangle inequality. While LLBs have been employed to answer kNN queries [18], this entailed significant pre-processing time and space. Specifically, the number of landmarks that can be chosen is limited due to high indexing cost, which leads to poor lower-bounds when applied to kNN search. Furthermore, there is no efficient data structure to compute minimum LLBs to groups of POIs to perform hierarchical search similar to R-trees and Euclidean distance.

1.2. Contributions

Motivated by the aforementioned under-served real-world POI search queries and by the gap in the existing work, our primary contribution is the **COL-Tree index**, an efficient data structure for hierarchical search in road networks to locate POIs by landmark lower-bounds. COL-Tree enables utilization of significantly more landmarks than existing works, while keeping the pre-processing cost small in theory and practice. Our further contributions are as follows:

- SUL-Trees: a supporting index that significantly accelerates COL-Tree construction with pre-processing cost theoretically and experimentally lower or comparable to popular road network indexes.

- Highly efficient heuristic search algorithms that utilize COL-Tree to answer AkNN, kFN, and range queries. We show COL-Trees are particularly suited for AkNN search due to a unique COL-Tree property that exploits the convexity-preserving nature of all common aggregate functions and for kFN search by using incremental candidate retrieval by upper-bounds for the first time.

- Our extensive experiments demonstrate the significant improvement in query time and heuristic efficiency of our techniques, achieving up to 4 orders of magnitude of improvement.

This manuscript is an extended version of a conference paper [21] selected as the best paper at the 30th International Conference on Automated Planning and Scheduling (ICAPS2020) and invited for submission as an extended journal paper. Compared to the conference version, we have: (a) proposed highly efficient query algorithms for two new queries (kFN and range queries); (b) reduced pre-processing time of SUL-Trees with a novel variant of Dijkstra search for subgraphs; (c) improved SUL-Tree index size and implementation

Symbol	Description
G	A road network graph
V	The vertex set of G, e.g., representing intersections
E	The edge set of G, e.g., representing road segments
P	The POI (object) vertex set ($P \subseteq V$)
k	The number of result objects a query must return ($k \leq |P|$)
$d(s,t)$	Shortest path (network) distance between s and t in G
$LB(s,t)$	A lower-bound on network distance, i.e., $LB(s,t) \leq d(s,t)$
$UB(s,t)$	An upper-bound on network distance, i.e., $UB(s,t) \geq d(s,t)$
$agg()$	A monotonic function that aggregates multiple input values, e.g., *sum* or *max*
ODL	List of object-landmark distances $(o, d(l,o)) \forall (o \in n_{leaf} \subseteq P)$ for COL-Tree leaf node n_{leaf}
SDL	List of subgraph vertex-landmark distances $(v, d(l,v)) \forall v \in n_T$ for SUL-Tree node n_T

Table 1: Summary of frequently used notations and symbols

efficiency by realizing a subgraph reordering technique that was proposed theoretically in the previous version; and (d) improved query performance with a root-based lower-bound optimization and exhaustive parameter testing (with new experiments and algorithmic insights on parameter choice). Note that we have renamed the data structures to COL-Tree and SUL-Tree from the previous version for clarity and easier pronunciation (apart from improved pre-processing, the data structures are unchanged).

2. Preliminaries

Road Network: We define a road network as a graph $G = (V, E)$. V is a vertex set and E is an edge set. Each edge $(u, v) \in E$ connects two vertices with weight $w(u, v)$ representing any real positive metric, e.g., the length or travel time of the edge. Network distance $d(s, t)$ is the minimum sum of weights connecting vertices s and t. We consider queries and objects (POIs) on graph vertices for simpler exposition.

Object Search Queries: Finding relevant objects according to some criteria on their network distance is an extremely common task in map-based services. As motivated in Section 1, we focus on solving the following important object search queries for a set of object vertices $P \subseteq V$:

1. **Aggregate k Nearest Neighbor (AkNN) Queries:** Given a set of query vertices $Q \subseteq V$ and an aggregation function agg, an AkNN query retrieves the k objects in P with minimum aggregate distances from set Q. Aggregate distance $d_{agg}(Q, p)$ to object p is computed by aggregating the network distances to p from each query vertex $q \in Q$ by function agg. In this paper, we focus on presenting techniques for the case when the agg function is either *sum* or *max*. We remark that AkNN queries are also known as Group kNN queries [22].

2. **k Farthest Neighbors (kFN) Queries:** Given a query vertex $q \in V$, a kFN query retrieves the k objects in P with maximum network distances from q.

3. **Range Queries:** Given a query vertex $q \in V$ and a distance threshold r, a range query retrieves *all* objects $p \in P$ that satisfy $d(q, p) \leq r$ (i.e., within the range).

Note that kNN queries are a special case of AkNN queries where Q contains only a single query location. For completeness we present results for kNN queries in Appendix A.

Landmark Lower-Bounds (LLBs): Landmark lower-bounds, also called differential heuristics [23], involve selecting a set L of m "landmark" vertices and then pre-computing distances from each landmark to all vertices in V. Given two vertices q and p, a lower-bound on network distance can be computed by Eq. (1) using the distances to landmark l_i and the triangle inequality. The maximum lower-bound over all m landmarks given by Eq. (2) gives the tightest lower-bound and is typically accurate even for small m [20].

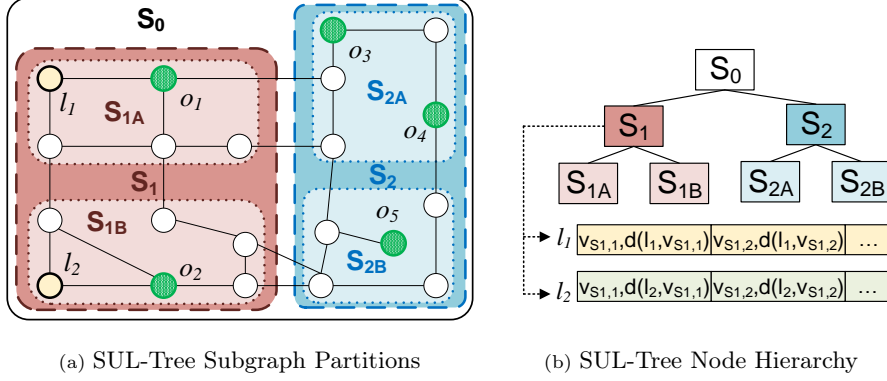$$LB_{l_i}(q, p) = |d(l_i, q) - d(l_i, p)| \leq d(q, p) \tag{1}$$

4

(a) SUL-Tree Subgraph Partitions   (b) SUL-Tree Node Hierarchy

Figure 1: Subgraph Landmark Tree (SUL-Tree)

$$LB_{max}(q, p) = \max_{l_i \in L}(|d(l_i, q) - d(l_i, p)|) \tag{2}$$

Landmark Upper-Bounds (LUBs):   Using the same pre-computed distances, an upper-bound can also be computed using the triangle inequality as in Eq. (3). In this case, the minimum upper-bound is chosen over all m landmarks to obtain the tightest upper-bound.

$$UB_{min}(q, p) = \min_{l_i \in L}(d(l_i, q) + d(l_i, p)| \tag{3}$$

Lower-Bound Aggregate Distances: Lemma 1 [4] shows, for monotonic aggregate functions, the aggregate of lower-bound distances to object p from each query vertex in Q is a lower-bound $LB_{agg}(Q, p)$ on aggregate distance $d_{agg}(Q, p)$.

Lemma 1. *Given a monotonic aggregate function agg and lower-bounds distance $LB(q_i, p)$ from each query vertex $q_i \in Q$ to object p, the aggregation of the lower-bound distances gives a lower-bound aggregate distance on the true aggregate distance. I.e., $LB_{agg}(Q, p) = agg(LB(q_0, p), \dots, LB(q_{|Q|}, p)) \leq d_{agg}(Q, p)$.*

3. Data Structures

Before introducing our main data structure, the Compacted Object Landmark Tree (COL-Tree), to enable efficient hierarchical object search in road networks, we first describe a supporting index called a Subgraph-Landmark Tree (SUL-Tree). SUL-Trees are not strictly necessary to build COL-Trees, but significantly accelerate their construction and also help introduce the hierarchical subgraph structure.

3.1. Road Network Index: Subgraph-Landmark Tree (SUL-Tree)

A SUL-Tree is an index for a given road network G. Each node in the SUL-Tree represents a subgraph of the road network with G being the tree's root. G is recursively partitioned into b disjoint subgraphs of equal size, stopping when a subgraph has no more than α vertices. Figure 1a shows such a partitioning for $b = 2$ and $\alpha = 6$ with the corresponding SUL-Tree shown in Figure 1b. For example, root S_0 is partitioned into subgraphs S_1 and S_2 in Figure 1a, which are represented as the children of S_0 in Figure 1b. Note that we refer to tree *nodes* and road network *vertices*.

For each tree node n_T, we select m of its vertices as *local landmarks*, e.g., l_1 and l_2 with $m = 2$ in S_1 in Figure 1a (landmarks for other nodes are omitted for clarity). We then compute a Subgraph Distance List (SDL) SDL_i for each landmark l_i of node n_T, containing the distances from l_i to all vertices contained in the subgraph associated with n_T. Figure 1b shows the SDLs for the landmarks of S_1 (those for other nodes are again omitted for clarity). SDL_i can be computed using multi-target Dijkstra search with the

landmark l_i as the source and all subgraph vertices as the targets. Note that the Dijkstra search must expand vertices irrespective of whether they are contained in n_T or not until all vertices of n_T are settled. This is to ensure correctness because shortest paths between l_i and some subgraph vertices may leave and then re-enter the subgraph. In Section 8.2, we present an optimized subgraph Dijkstra that restricts this search using landmarks in higher levels of the SUL-Tree. Furthermore, subgraph vertices are stored implicitly by vertex ID ranges using a subgraph vertex ordering presented in Section 8.1 (along with other space optimizations). Any graph partitioning method that can output similar-sized subgraphs can be used, e.g., we utilize the widely used METIS [24].

3.2. Object Index: Compacted Object Landmark Tree (COL-Tree)

The SUL-Tree can be used to efficiently construct our object index, the Compacted Object-Landmark Tree (COL-Tree). COL-Tree is a carefully compacted version of the SUL-Tree for object set P. Compaction ensures that there are m local landmarks for at most λ objects in leaf nodes, to increase the likelihood of finding a tighter lower-bound for more objects. Note that the SUL-Tree is shared between initial construction and any rebuilding of all COL-Trees, i.e., for many different object sets and potentially multiple times.

Given SUL-Tree T and object set P, a COL-Tree C is constructed by visiting nodes in T in a top-down manner and creating corresponding nodes in C. Let n_T be the currently visited node in T (initially the root). A corresponding node n_C is created in C for n_T. Let λ be the maximum number of objects allowed in a leaf node of a COL-Tree. If n_T contains more than λ objects, the search expands to its children. Otherwise, the construction is pruned at n_C, which becomes a leaf-node of C. For the new leaf n_C, an *Object Distance List* (ODL) ODL_i is created in n_C for each landmark l_i in n_T. These are simply the SDLs of n_T, except with only the distances for object vertices from P. Figure 2a shows a COL-Tree index for $\lambda = 2$ constructed from the SUL-Tree in Figure 1b based on the 5 object vertices (green shaded vertices) in Figure 1a. Note that S_{1A} and S_{1B} were removed as construction was pruned at S_1 due to its number of objects (the $ODLs$ of other nodes are omitted for clarity). Also note that we have chosen $\lambda < \alpha$ to simplify the example, but generally require $\lambda \geq \alpha$ to guarantee no leaf node and therefore no ODL contains more than λ objects. For example, if every vertex in a leaf node is an object there will be α objects in its ODL.

Each ODL_i of a leaf node n_{leaf} in C is sorted by distance. In non-leaf nodes n_C, we only store the minimum distance min_{n_C,l_i} and maximum distance max_{n_C,l_i} to any object in the node from each landmark l_i. These can be computed by lookup of the Subgraph Distance List SDL_i of the SUL-Tree node corresponding to n_C. As we describe next, these bookkeeping components of COL-Tree allow computation of lower-bounds to nodes and traversal of the hierarchy.
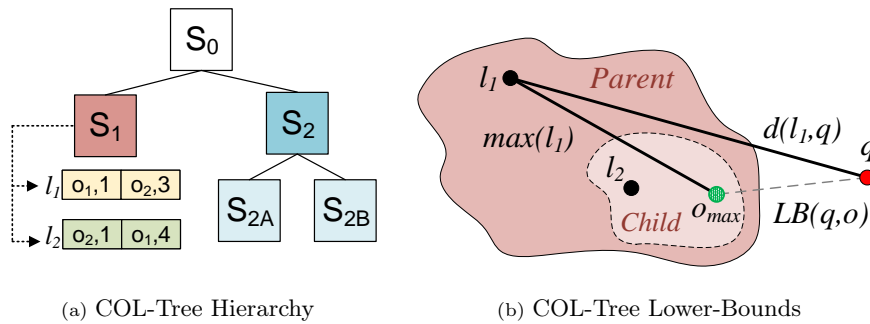

(a) COL-Tree Hierarchy

(b) COL-Tree Lower-Bounds

Figure 2: Compacted Object-Landmark Tree (COL-Tree)

3.3. Lower-Bound Heuristic for Graph Traversal

Similar to Eq. (1), we can compute a lower bound for all objects contained within a node n_C in COL-Tree C by Eq. (4) for one landmark l_i of n_C. Eq. (5) gives the best lower-bound over all m landmarks of n_C:

$$LB_{l_i}(q, n_C) = \begin{cases} d(l_i, q) - M^+ & \text{if } d(l_i, q) \geq M^+ \\ M^- - d(l_i, q) & \text{if } d(l_i, q) \leq M^- \\ 0 & \text{else} \end{cases} \tag{4}$$

$$LB_{max}(q, n_C) = \max_{l_i \in n_C} (LB_{l_i}(q, n_C)) \tag{5}$$

Similar to Eq. (3), we can also compute an upper-bound on the distance to any object by Eq. (6). This will be highly useful for efficiently computing kFNs and other queries based on farthest metrics.

$$UB_{min}(q, n_C) = \min_{l_i \in n_C} (d(l_i, q) + M^+) \tag{6}$$

Network Distance Substitution: $M^- := min_{n_C, l_i}$ and $M^+ := max_{n_C, l_i}$ for n_C are already available in COL-Tree. However, for non-root nodes, $d(l_i, q)$ in Eq. (4) is only available if l_i and q are in the same subgraph. Pre-computing this distance for all V and landmarks is infeasible given the space implications and defeats the purpose of using subgraph landmarks. Alternatively, computing $d(l_i, q)$ on the fly during query time is expensive and may be wasteful if the node does not contain results. Clearly, Eq. (4) still holds if we replace the distances with lower-bound $LB(l_i, q)$ and upper bound $UB(l_i, q)$, as in Eq. (7). The SDLs of the SUL-Tree root or lowest common ancestor node can be conveniently used to compute the best $LB(l_i, q)$ and $UB(l_i, q)$ by Eq. (2) and its upper-bound by Eq. (3), respectively. Choosing the tightest over all landmarks of n_C gives an inexpensive and accurate bound even for a small number of landmarks:

$$LB_{l_i}(q, n_C) = \begin{cases} LB(l_i, q) - M^+ & \text{if } LB(l_i, q) \geq M^+ \\ M^- - UB(l_i, q) & \text{if } UB(l_i, q) \leq M^- \\ 0 & \text{else} \end{cases} \tag{7}$$

Alternate Bound by Root Landmarks: Another cheaper alternative to Eq. (4) can be computed using the root landmarks. During COL-Tree construction, in addition to M^- and M^+ for the node's landmarks, we can also compute the minimum and maximum distances, M_R^- and M_R^+, from the root landmarks to any object essentially for free. This is because SDL_i of SUL-Tree root node n_R contains the distances from each root landmark l_i to all vertices. Thus, during querying, we can similarly retrieve $d(q, l_i)$ in $O(1)$ time for each root landmark by looking up SDL_i of n_R and utilize it in Eq. (4) with M_R^- and M_R^+ to compute an alternate lower-bound. We can simply choose the tighter value between this and Eq. (7). The alternate bound is likely to help when $LB(l_i, q)$ or $UB(l_i, q)$ in Eq. (7) are less accurate.

3.3.1. SUL-Tree Landmark Selection Policy

Landmark vertices must be chosen during SUL-Tree construction such that they produce good lower-bounds irrespective of object set P. Past studies have shown that randomly chosen landmarks work reasonably well in-practice [20], which we also find to be true. However, as observed in past work, some landmark selection policies may result in better lower-bounds. For example, Goldberg and Harrelson [20] showed that landmarks at fringes of the graph resulted in better lower-bounds because it increased the likelihood of a source vertex being an ancestor (or having less deviation from the ancestor) of a target vertex in the shortest path tree from the landmark. Inspired by their approaches, we propose several landmark selection policies, which we evaluate in our experiments against a random baseline:

- *Furthest Border*: Inspired by the *furthest vertex* selection policy of [20], we adapt this definition for subgraphs. Starting with a randomly chosen border vertex, the remaining border that is furthest from the landmarks chosen thus far is selected as the next landmark. If there are insufficient borders to select m landmarks, random subgraph vertices are chosen.

- *Slice-Based Furthest Border*: We first partition the plane into m equally sized slices around the Euclidean center of the subgraph associated with the SUL-Tree node, as if we were cutting a cake. We

choose a border as a landmark from each slice. If a slice has no borders, we choose the slice vertex furthest from the Euclidean center. If a slice has no vertices, we randomly choose a subgraph vertex to ensure m landmarks. The intuition for this method is that there is always a landmark "behind" the objects and a query vertex, which is more likely to produce better lower-bounds.

- *Border Minmax*: Existing work on landmark selection has focused on improving point-to-point shortest path queries, but their intuitions do not necessarily translate to object search. For example, a landmark "behind" the objects may only produce a good lower-bound for one query vertex, which is unlikely to help AkNN queries with multiple query vertices. To better accommodate such scenarios, we propose the *Border Minmax* policy, which attempts to find an approximate subgraph vertex center. After randomly choosing a small subset of subgraph borders, we compute the maximum distance from each subgraph vertex to these borders (performed efficiently via Dijkstra search on the reverse graph). We choose the m subgraph vertices with minimum maximum distances to the subset of borders.

3.3.2. Tightening Lower-Bounds with Tree Depth

The accuracy of COL-Tree's inexpensive lower-bounds increases as we delve deeper into the hierarchy. In Figure 2a, let us say we use l_1 and its maximum object distance to compute a lower-bound for the parent node. At the lower level, we may use the child's landmarks like l_2, which are local to the objects and more likely to produce a better lower-bound. This lets us differentiate tree branches and pinpoint the most promising candidates. Finally, at the leaf nodes (with objects), COL-Tree guarantees there are at most λ objects per landmark to maximize differentiation.

4. Query Algorithm for AkNN Search

The *decoupled heuristic* paradigm is a highly effective strategy for POI search in road networks [25, 17]. This involves iteratively retrieving candidates objects (POIs) by an admissible heuristic and using a separate road network index to compute exact distances. A set of the best candidates found so far is maintained with termination occurring at optimality when no remaining object can improve the result set. The exact cost computation is "decoupled" from the search heuristic allowing any road network index to be used, which have been extensively studied for decades [26, 27]. Here we show that COL-Trees are ideally suited to answer AkNN queries using this paradigm. Specifically, COL-Tree considers fewer AkNN candidates to find the object with the minimum lower-bound using a novel property of ODLs in leaf nodes.

4.1. Object Distance Lists and Convexity

Section 3.3 described how to efficiently navigate a COL-Tree to find the leaf node that likely contains the next best candidate object by lower-bound distance. Then, retrieving candidate objects from the leaf node n_{leaf} involves carefully evaluating its ODLs. Note that Eq. (1) can be expressed as an absolute value function of form $f(x) = |C - x|$ for some landmark l. Here, C is the constant distance $d(l, q)$ between the landmark l and the query point q, and x is a variable distance $d(l, p)$ depending on the object $p \in P$. Since absolute-value functions are convex, $f(x)$ is minimized for x closest to C. This property is useful to find the minimum lower-bound in an ODL_l for landmark l for a single query vertex q. Since ODL_l essentially stores the domain of x (i.e., $d(l_i, o) \forall o \in n_{leaf}$), the minimum lower-bound for the landmark l can be found by searching ODL_l for $d(l, c)$ closest to $d(l, q)$ for some object $c \in ODL_l$. Since ODL_l is sorted, this is possible using a modified binary search, as observed by [18], in only $O(\log \lambda)$ time.

It is not immediately clear whether this idea can be extended to minimum lower-bound *aggregate* distance, which is complicated by the presence of multiple query locations $q_i \in Q$. From Lemma 1, we know the aggregate of *lower-bound* distances from each query vertex q_i is a lower-bound on aggregate distance for monotonic functions [4]. Therefore, the function to minimize becomes $f(x) = agg(|C_1 - x|, \cdots, |C_n - x|)$ for a monotonic aggregate function agg where C_i is $d(l, q_i)$ for the given landmark l and a query $q_i \in Q$. At first glance, it appears we need to search ODLs for multiple values (i.e., for each C_i) to find the object with minimum aggregate lower-bound. Surprisingly, it is not necessary for aggregate functions that preserve

convexity. Moreover, most widely used functions [28], such as *max* and *sum*, do preserve convexity. Boyd and Vandenberghe [29] prove convexity preservation for a wide range of functions.

Specifically, once the minimizing x^* for the function $f(x)$ is found, iteratively retrieving the object that gives the next smallest lower-bound simply requires checking the element to the right or left of x^* in the ODL, due to the convexity of the function. This allows us to incrementally retrieve the objects by their lower-bounds and enable termination at optimality. However, unlike the single query case, finding the minimum of $f(x)$ is not obvious for aggregate kNN queries. Below, we show how to find the minimum for the two most common aggregate functions, *max* and *sum*.

Lemma 2. *Consider the aggregate function defined by the sum of a set of absolute functions* $f(x) = sum(|C_1 - x|, \cdots, |C_n - x|)$*. The minimum* x^* *of* $f(x)$ *is the median value of the constants* $C_1, ..., C_n$*.*

Proof. Let constants be sorted such that $C_1 \leq C_2 \leq ... \leq C_n$. Let x^* be the median of these constants. We show that $f(x^*) \leq f(x')$ for all x'. Let $d = |x^* - x'|$. Without loss of generality, assume $x' < x^*$. For each $C_k < x^*$, the difference between $|C_k - x^*|$ and $|C_k - x'|$ is at most d, i.e., $|C_k - x^*| - |C_k - x'| \leq d$. On the other hand, for each $C_j \geq x^*$, the difference between $|C_j - x'|$ and $|C_j - x^*|$ is exactly d, i.e., $|C_j - x'| - |C_j - x^*| = d$. Since x^* is median of the constants, the number of constants $C_j \geq x^*$ is at least $\lceil \frac{n}{2} \rceil$. In other words, for at least half of the constants, $|C_i - x'| - |C_i - x^*| = d$ and for each of the remaining constants, $|C_i - x^*| - |C_i - x'| \leq d$. Thus, $f(x^*) \leq f(x')$. □

Lemma 3. *Consider the aggregate function defined by the maximum of a set of absolute functions* $f(x) = max(|C_1 - x|, \cdots, |C_n - x|)$*. The minimum* x^* *of* $f(x)$ *is* $\frac{C_{min} + C_{max}}{2}$*, i.e., the average of the minimum and maximum constants.*

Proof. For sorted constants, let $C_1 = C_{min}$ and $C_n = C_{max}$. Note that $f(x) = max(|C_1 - x|, \cdots, |C_n - x|) = max(|C_1 - x|, |C_n - x|)$. Thus, the minimum value of $f(x)$ is $\frac{C_1 + C_n}{2}$, i.e., minimum x^* of $f(x)$ is $\frac{C_{min} + C_{max}}{2}$. □

4.2. Search Algorithm

Algorithm 1 uses hierarchical traversal of the COL-Tree by node lower-bounds to guide us towards ODLs in leaf nodes most likely to contain AkNN results. While SUL-Trees are an input to Algorithm 1, only the root level is needed to compute lower and upper-bounds for Eq. (7). The algorithm maintains a priority queue $\mathcal{PQ}$ containing objects and COL-Tree nodes keyed by their aggregate lower-bound distances from Q. The loop iteratively extracts the minimum lower-bound queue element. If an object is extracted, its exact aggregate distance is computed and the result set R and D_k are updated (lines 6 to 9). The result set R maintains up to k objects with the smallest aggregate distances found so far by the algorithm and D_k corresponds to the k-th largest aggregate distance among these objects.

If a non-leaf node is extracted then an aggregate lower-bound score is computed according to Eq. (5) and Eq. (7) for each of its child nodes (lines 11 to 13). If it is a leaf-node, it is initialized if encountered for the first time (lines 15 to 19). An ODL to process is chosen (Line 16), the constants in the absolute value functions are computed, and a binary search is performed to find the minimizing list index given by Lemma 2 or 3. Pointers RP and LP are initialized to the index of the minimizing value. Then objects are retrieved from the list using RETRIEVEOBJECTSODL and, if the list is not completely searched, the node is re-inserted into $\mathcal{PQ}$ with minimum lower-bound computed by RP or LP. $LB_{agg,max}(Q,p)$ on Line 29 computes the maximum (best) lower-bound aggregate distance for object p over all landmarks for which networks distances to p are already computed (e.g., landmarks in the root SUL-Tree node or $c.l$ in the leaf). We use the lower-bound for p to avoid unnecessary network distance computations (if p is a result it will be processed at Line 8). Note that $LB_{agg,c.l}(Q,c)$ on Line 25 refers to the lower-bound for all yet-to-be retrieved objects in the ODL of leaf node c corresponding to landmark $c.l$ (i.e., subject to $c.RP$ and $c.LP$).

Recall that leaf nodes contain ODLs for each of its m landmarks so that we can choose the ODL most likely to support the best candidate retrieval. For kNN queries, the landmark furthest from the query vertex is likely to be "behind" more objects and thereby result in better lower-bounds for more objects

Algorithm 1 Get AkNNs by COL-Tree for query vertex set Q

1: **function** GETAKNNS($k, Q, agg, COL\text{-}Tree, SUL\text{-}Tree$)
2: Initialize min queue $\mathcal{PQ} \leftarrow \phi$ and result set (max queue) $R \leftarrow \phi$
3: INSERT($\mathcal{PQ}, COL\text{-}Tree.root, 0$) , $D_k \leftarrow \infty$
4: **while** MINKEY($\mathcal{PQ}$) $< D_k$ and $\mathcal{PQ}$ not empty **do**
5:   $c \leftarrow$ EXTRACT-MIN($\mathcal{PQ}$)
6:   **if** c is an object **then**
7:    Compute $d(q_i, c) \forall q_i \in Q$ then apply agg to get $d_{agg}(Q, c)$
8:    **if** $d_{agg}(Q, c) < D_k$ **then**
9:     INSERT($R, c, d_{agg}(Q, c)$), EXTRACT-MAX(R), $D_k \leftarrow$ MAXKEY(R)
10:   **else if** c is a non-leaf node **then**
11:    **for each** child node e of c **do**
12:     Compute best $LB_{agg}(Q, e)$ by Lemma 1 and node lower-bounds in Section 3.3
13:     INSERT($\mathcal{PQ}, e, LB_{agg,max}(Q, e)$)
14:   **else if** c is a leaf node **then**
15:    **if** c not seen before **then**
16:     Choose $ODL_{c.l}$ by landmark $c.l$ in c with max average distance from Q by LB
17:     Compute $d(q, c.l) \forall q \in Q$ and save for reuse
18:     Binary search element i in $ODL_{c.l}$ with min $LB_{agg}(Q, c)$ by Lemma 2 or 3
19:     Let $c.RP \leftarrow i$ and $c.LP \leftarrow i$
20:    RETRIEVEOBJECTSODL($\mathcal{PQ}, Q, c$)
21:    Compute $LB_{agg}(Q, c)$ for remaining $ODL_{c.l}$ objects (will be by $c.RP$ or $c.LP$)
22:    INSERT($\mathcal{PQ}, c, LB_{agg}(Q, c)$)
23: **return** R
24: **function** RETRIEVEOBJECTSODL($\mathcal{PQ}, Q, c$)
25: **while** $min(LB_{agg}(Q, ODL_{c.l}[c.LP]), LB_{agg}(Q, ODL_{c.l}[c.RP])) \leq$ MINKEY($\mathcal{PQ}$) **do**
26:   $p \leftarrow$ object at $c.LP$ or $c.RP$ with smaller lower-bound by $ODL_c.l$
27:   Increment $c.RP$ or decrement $c.LP$ used to set p at Line 26
28:   Compute best $LB_{agg}(Q, p)$ by $c.l$ or $SUL\text{-}Tree$ root landmarks
29:   INSERT($\mathcal{PQ}, p, LB_{agg,max}(Q, p)$)

(Section 3.3.1). Since AkNN involve multiple query vertices, at Line 16, we choose $c.l$ as the landmark with the largest average distance from the query vertices Q. Since we compute the lower-bound distance from query vertices to each landmark during node lower-bound computation using the SUL-Tree root's SDLs (Line 12), for leaf nodes we can also simply keep track of the landmark with largest average lower-bound distance. This provides an approximation on the furthest landmark essentially for free. Note ODL selection does not affect correctness, but some ODLs may generate fewer false candidates.

Proof Sketch for Correctness: The intuition behind the correctness of Algorithm 1 is that the hierarchical tree search maintains a lower-bound aggregate distance for every unseen object in the priority queue at all times. That is, there is a queue element for either (a) the object itself; or (b) a tree node (i.e., subgraph) containing the object. Thus, when the algorithm terminates, lower-bounds for all remaining objects are greater than D_k, the true aggregate distance for the k result objects found so far. In other words, no other object can have an aggregate distance smaller than the current k result objects. Non-result objects extracted from the queue (because the lower-bound was less than D_k) whose exact distance is greater than D_k cannot improve the current result set and can be safely discarded.

5. Query Algorithm for kFN Search

Euclidean k Farthest Neighbors search can be performed through a simple variation on kNN search. For example, kNN search can be conducted using a lower-bound to all objects contained in an R-tree node by computing the minimum distance to its MBR. An upper-bound can be computed with similar ease, allowing Euclidean kFN algorithms to retrieve candidate objects by upper-bound and terminate when the next best (i.e., largest) upper-bound cannot improve the result set. Papadias et al. [2] observed that Euclidean distance can be used as a lower-bound for kNN search in road networks. Their technique, IER, was later shown to perform extremely well even when edge weights were travel-time [25] (a lower-bound being obtained by dividing Euclidean distance by the maximum speed in the road network).

However, Euclidean distance *cannot* similarly be used for kFN search in road networks. Although an upper-bound on the farthest Euclidean distance to an object can be obtained, the Euclidean distance can still only be used to compute a *lower-bound* on network distance. Therefore, it cannot readily be used for kFN search in road networks as it is not admissible for this maximization problem. Moreover, there are no efficient data structures that could support road network kFN search without evaluating all objects. An expansion-based approach would involve incrementally retrieving all objects until the farthest one is reached, meaning it is more efficient to simply evaluate all objects in a loop. Fortunately, COL-Trees can support incremental retrieval of the best candidate by *upper-bound*, as we demonstrate next.

5.1. Search Algorithm

In contrast to AkNN search, kFN search in Algorithm 2 relies on maximum priority queue $\mathcal{PQ}$ keyed by the upper-bound distance to tree nodes and objects from the query q to guide the hierarchical search. As in Algorithm 1, priority queue elements that are objects, non-leaf nodes, and are handled separately but adapted for kFN search. For example, at Line 12, we utilize Eq. (6) to compute an upper-bound distance to all objects contained in a COL-Tree node. Another difference is that the best candidate object can be initialized in $O(1)$ time without a binary search as the final element in the ODL (Line 18) and then we need only keep track of $c.RP$ (Line 25), the right-most element of the ODL evaluated so far. This is because the ODL is sorted in increasing order of distance from the landmark and the object with largest distance from the landmark will clearly give the maximum (i.e., best) upper-bound by Eq. (3).

The last major point of difference is that we choose the ODL associated with the landmark closest to the query, as it is more likely to lie on or close to the shortest path from the query vertex to the objects leading to more accurate upper-bounds [30]. An approximation on this landmark can be conveniently saved during node upper-bound computation for leaf nodes (Line 12) using SUL-Tree root SDLs, by keeping track of the landmark with the smallest upper-bound distance to q. Algorithm 2 retrieves the object with the next best (maximum) upper-bound until the results set R cannot be improved, when it terminates returning the k objects with the largest distances from q.

Note that $UB_{c.l}(q, c)$ on Line 24 refers to the upper-bound for all yet-to-be retrieved objects in the ODL of leaf node c corresponding to landmark $c.l$ (i.e., subject to $c.RP$).

Algorithm 2 Get kFNs by COL-Tree for query vertex q

1: **function** GETKFNS$(k, q, COL\text{-}Tree, SUL\text{-}Tree)$
2:     Initialize max queue $\mathcal{PQ} \leftarrow \phi$ and result set (min queue) $R \leftarrow \phi$
3:     INSERT$(\mathcal{PQ}, COL\text{-}Tree.root, 0)$, $D_k \leftarrow 0$
4:     **while** MAXKEY$(\mathcal{PQ}) > D_k$ and $\mathcal{PQ}$ not empty **do**
5:         $c \leftarrow$ EXTRACT-MAX$(\mathcal{PQ})$
6:         **if** c is an object **then**
7:             Compute $d(q, c)$
8:             **if** $d(q, c) > D_k$ **then**
9:                 INSERT$(R, c, d(q, c))$, EXTRACT-MIN(R), $D_k \leftarrow$ MINKEY(R)
10:         **else if** c is a non-leaf node **then**
11:             **for each** child node e of c **do**
12:                 Compute best $UB(q, e)$ by node upper-bounds in Section 3.3
13:                 INSERT$(\mathcal{PQ}, e, UB_{min}(q, e))$
14:         **else if** c is a leaf node **then**
15:             **if** c not seen before **then**
16:                 Choose $ODL_{c.l}$ by landmark $c.l$ in c closest to q by UB
17:                 Compute $d(q, c.l)$ and save for reuse
18:                 Initialize $c.RP$ to final element in $ODL_{c.l}$
19:             RETRIEVEOBJECTSODL$(\mathcal{PQ}, q, c)$
20:             Compute $UB(q, c)$ for remaining $ODL_{c.l}$ objects by $c.RP$
21:             INSERT$(\mathcal{PQ}, c, UB(q, c))$
22:     **return** R
23: **function** RETRIEVEOBJECTSODL$(\mathcal{PQ}, q, c)$
24:     **while** $UB_{c.l}(q, ODL_{c.l}[c.RP]) \geq$ MAXKEY$(\mathcal{PQ})$ **do**
25:         $p \leftarrow$ object at $c.RP$ and decrement $c.RP$
26:         Compute best $UB(q, p)$ by $c.l$ or $SUL\text{-}Tree$ root landmarks
27:         INSERT$(\mathcal{PQ}, p, UB_{min}(q, p))$

---

6. Query Algorithm for Range Search

While the benefit of COL-Tree's hierarchical approach can be clearly intuited in the case of AkNN and kFN queries, whether this translates to purely proximity-based search problems such as kNN and range queries is less clear. The state-of-the-art kNN search heuristic [18] utilizes an adjacency property of Network Voronoi Diagrams (NVD) to incrementally retrieve the next nearest object from a pool of candidates likely to contain it. Such greedy incremental approaches translate well for kNN queries, where our goal is to find the k results and terminate as soon as possible. While range queries also retrieve objects by proximity, they do not have the benefit of stopping after a fixed number of results, instead retrieving all objects within a radius r. Since COL-Tree can be used to compute lower- and upper-bounds to all objects in entire subgraphs, this can be leveraged to determine whether multiple objects are entirely within or outside radius r in the best case. A COL-Tree approach is further motivated by the significant $O(|V| \log |V|)$ pre-processing time of NVDs [18] compared to the $O(m|P| \log |P|)$ of COL-Trees (Table 2). In this section we present an algorithm to use COL-Trees to efficiently answer range queries based on network distance.

6.1. Search Algorithm

Algorithm 3 describes how to use COL-Tree for range queries. Unlike all competing methods, COL-Tree range search *does not* require a priority queue, and exploration can be implemented using a stack with reduced insertion and extraction costs (Line 2). Note that either DFS or BFS of the COL-Tree can be used

as both will visit the same nodes. If the node c popped from the stack is a non-leaf node, we evaluate each of its child nodes by computing their lower- and upper-bounds. If the upper-bound is less than or equal to the radius r (Line 8), then all objects must be results. An inner stack can be used to conduct a DFS from c to immediately retrieve all objects contained in the leaf node descendants of c (note that c may be a leaf node) and insert them in result set R (Line 9). However, c is pushed to the stack for further exploration if its upper-bound is greater than r and the lower-bound is less than or equal to r (Line 11), since it may have results. c is implicitly pruned if the lower-bound is greater than r as it cannot have results. Alternatively, if c is a leaf node, we evaluate its objects using RetrieveObjectsODL. First we select the $ODL_{c.l}$ to search (Line 16). From kFN queries we know that $ODL_{c.l}$ is, by definition, also sorted on upper-bound. Thus in Line 17, we can iterate over $ODL_{c.l}$ in increasing order of upper-bound and report all objects with upper-bound within radius r as results. Lines 19 to 23 retrieves any remaining results. Similar to AkNN search, the $ODL_{c.l}$ element associated with the object with minimum lower-bound can be found by binary search, allowing incremental lower-bound search to the left and right of this element. We account for objects already evaluated by upper-bound using IP set to the element at which the upper-bound based iteration stopped (i.e., all elements before this have already been added to the result). Note that at Lines 21 and 23, inexpensive lower- and upper-bounds computed using SUL-Tree root landmarks can be used to further filter objects and potentially avoid expensive network distance computations.

To elaborate on $ODL_{c.l}$ choice at Line 16, consider $M_{c.l}^- + UB(q, c.l)$, a variation on Eq. (6) giving the minimum upper-bound for any object in $ODL_{c.l}$. We choose $ODL_{c.l}$ with either: (1) smallest $M_{c.l}^- + UB(q, c.l) \leq r$; or (2) if all $M_{c.l}^- + UB(q, c.l) > r$ we choose the $ODL_{c.l}$ associated with the landmark of c furthest from q. Case (1) maximizes the number of objects that can be immediately determined as results by the loop at Line 17. In case (2) no object can be detected by upper-bound, so we choose the furthest landmark to increase the quality of lower-bounds. Similar to other queries, these landmarks can be approximated essentially for free during leaf node bound computation (Line 7).

Algorithm 3 Get objects within radius r from query vertex q by COL-Tree

---

1: **function** RangeSearch($r, q, COL\text{-}Tree, SUL\text{-}Tree$)
2:     Initialize stack $\mathcal{S} \leftarrow COL\text{-}Tree.root$ and result set $R \leftarrow \phi$
3:     **while** $\mathcal{S}$ not empty **do**
4:         $c \leftarrow$ Pop($\mathcal{S}$)
5:         **if** c is a non-leaf node **then**
6:           **for each** child node e of c **do**
7:             Compute best $UB(q, e)$ and $LB(q, e)$) by node bounds in Section 3.3
8:             **if** $UB_{min}(q, e) \leq r$ **then**
9:               Retrieve all objects $p \in e$ by DFS and insert into R
10:            **else if** $LB_{max}(q, e) \leq r$ **then**
11:               Push($\mathcal{S}, e$)
12:         **else if** c is a leaf node **then**
13:           RetrieveObjectsODL($\mathcal{S}, q, c$)
14:     **return** R
15: **function** RetrieveObjectsODL($\mathcal{S}, q, c$)
16:     Choose $ODL_{c.l}$ with smallest $M_{c.l}^- + UB(q, c.l) \leq r$ or landmark furthest from q by LB
17:     **for each** objects p of $ODL_{c.l}$ with $UB_{c.l}(q, p) \leq r$ **do** insert p into R
18:     Let IP be index in $ODL_{c.l}$ of first p with $UB_{c.l}(q, p) > r$
19:     Let LP be index for p with $max(d(c.l, p)) \leq d(q, c.l)$ via binary search and $RP = max(IP, LP + 1)$
20:     **while** $LB_{c.l}(q, ODL_{c.l}[LP]) \leq r$ and $LP \geq IP$ **do**
21:         Compute $d(q, ODL_{c.l}[LP])$, insert into R if $d(q, ODL_{c.l}[LP]) \leq r$, and decrement LP
22:     **while** $LB_{c.l}(q, ODL_{c.l}[RP]) \leq r$ **do**
23:         Compute $d(q, ODL_{c.l}[RP])$, insert into R if $d(q, ODL_{c.l}[RP]) \leq r$, and increment RP

---

7. Complexity Analysis

SUL-Trees and COL-Trees enable hierarchical search at the expense of increased the pre-processing costs over existing landmark-based indexes such as ALT (which, in any case, cannot support efficient object search). Although additional pre-processing cost is amortized with faster querying, we attempt to minimize this cost as much as possible, e.g., by limiting tree depth with λ. In this section we analyze the time and space complexity to show that increased pre-processing cost is small or comparable.

COL-Tree: A COL-Tree is created from a SUL-Tree, which is a complete b-ary tree. In a COL-Tree for object set P, there are at most $O(\frac{|P|}{\lambda})$ leaf nodes. The top-down conversion of the SUL-Tree may result in the COL-Tree being unbalanced as objects may be disproportionately distributed in SUL-Tree nodes. But during the conversion, empty SUL-Tree nodes are pruned, resulting in the total number of nodes in COL-Tree being the same as a complete b-ary tree, i.e., $O(\frac{|P|}{\lambda})$ (branching factor b is a small constant). However, COL-Tree's space complexity is dominated by the m Object Distance Lists (ODLs) of length λ stored in each leaf node. This results in $O(m|P|)$ total space as each object is only stored in one ODL. Compression of COL-Tree's height via implicit merging of sibling-less child nodes into parent nodes results in an average COL-Tree depth of $O(\log|P|)$. Given the top-down conversion and $O(1)$ look-ups (e.g., if implemented as hash-tables) of SUL-Tree Subgraph Distance Lists (SDLs) to construct ODLs, propagating $|P|$ objects to build m ODLs and computing M^-/M^+ takes $O(m|P|\log|P|)$ time. Sorting all ODLs takes $O(|P|\lambda\log\lambda)$ time where λ is a small constant. Thus, the total time complexity is $O(m|P|\log|P|)$. Note that m, the number landmarks per node, is a small constant.

SUL-Tree: The height of the balanced SUL-Tree, with branching factor b and $\frac{|V|}{\alpha}$ leaf nodes, is $O(\log_b \frac{|V|}{\alpha})$, i.e., $O(\log|V|)$ since b and α are small constants. Given the disjoint partitioning, each vertex belongs to at most one tree node at each level, meaning there are $O(|V|)$ vertices per level. Then all SDLs associated with the SUL-Tree will take $O(m|V|\log|V|)$ space, since we store the distance from every vertex to m landmarks at each level. This, of course, dominates the $O(|V|)$ space occupied by the tree nodes. For level i, there will be at most b^i nodes, each with $\frac{|V|}{b^i}$ vertices. If a Dijkstra's search from each of the m landmarks is used to compute the distances to the node's vertices, then this may take $O(b^i m \frac{|V|}{b^i} \log \frac{|V|}{b^i})$ time for all nodes at level i. However, the Dijkstra's search may visit vertices outside the SUL-Tree node's subgraph, e.g., if a shortest path from a landmark to some subgraph vertex passes outside the subgraph. We consider $\gamma \times \frac{|V|}{b^i}$ to represent the total number of vertices visited by the Dijkstra search. We find that γ is a small constant in practice (and can be further reduced as shown in Section 8). Multiplying by the height, the time complexity for the whole tree is $O(m|V|\log^2|V|)$.

Complexity	ALT [20]	SUL-Tree (Ours)	OL [18]	COL-Tree (Ours)
Space	$O(m|V|)$	$O(m|V|\log|V|)$	$O(m|P|)$	$O(m|P|)$
Time	$O(m|V|\log|V|)$	$O(m|V|\log^2|V|)$	$O(m|P|\log|P|)$	$O(m|P|\log|P|)$

Table 2: Comparison of space and time complexity for the proposed methods versus previous landmark-based indexes

Table 2 shows the theoretical space and time complexity of our techniques compare favorably with existing landmark-based road network and object indexes, ALT and Object Lists (OL)[1] (resp.), which do not support hierarchical object search. COL-Tree offers the same space and time as OL, while SUL-Tree's space and time cost only increases by a factor of $O(\log|V|)$ over ALT.

8. Pre-Processing Optimizations

8.1. Subgraph Vertex Ordering

In Section 3.1, the top-down conversion of a SUL-Tree to a COL-Tree for object set P requires the root-to-leaf propagation of each object $p \in P$. To do this efficiently, we must determine the child node p belongs

---

[1]We derive the time complexity of OL as it was not listed in [18] (which is dominated by sorting m lists of $|P|$ objects)

to in $O(1)$ time. We must also retrieve the distance from each landmark to p from the SDL of each relevant SUL-Tree node in $O(1)$ time. However, in practice, achieving $O(1)$ containment and distance retrieval has unfavorable implications for memory (both in terms of total usage and access latency). First, this would necessarily require storing the same vertex at each level of the SUL-Tree (i.e., each node/subgraph a vertex is associated with). Second, $O(1)$ operations require using highly memory inefficient data structures like hash-tables. While hash-tables are an elegant theoretical construct, actual implementation of hash-tables such as `std::unordered_map` in C++ take up significantly more memory than just its elements. Hash-tables also involve increased memory latency [25], which can often significantly reduce algorithm performance [31]. To avoid such issues one could attempt to only store vertices in the leaf node of the SUL-Tree. With only implicit storage of vertices in non-leaf nodes, the propagation of object $p \in P$ would need to be performed in a bottom-up manner taking $O(log|V|)$ time. This would drastically affect COL-Tree construction time (Section 7), making it dependent on $|V|$ rather than the generally far smaller $|P|$.

However, these issues can be entirely avoided by using a novel *subgraph vertex ordering* which groups vertices of child subgraphs recursively as shown in Figure 3 and numbering vertex IDs based on this order. By simply storing the first and last ID of vertices contained by a SUL-Tree node similar to an interval tree, we can achieve the necessary $O(1)$ complexity operations without resorting to highly memory inefficient data structures like hash-tables, which is problematic as discussed above. In fact, as an added benefit of the subgraph vertex ordering, this significantly reduces the space cost of the SUL-Tree (i.e., more than just memory usage). The subgraph vertex IDs do not need to be stored *at all* (the vertex IDs can be inferred from the SDL index). The distances can simply be stored in an array in vertex ID order with access to the distance for a specific vertex being achieved through simple arithmetic using the first vertex ID of the containing node. Although it does not change the complexity, for large-scale road networks, this can reduce real-world index size by gigabytes. We evaluate all improvements experimentally in Section 9.5.1.
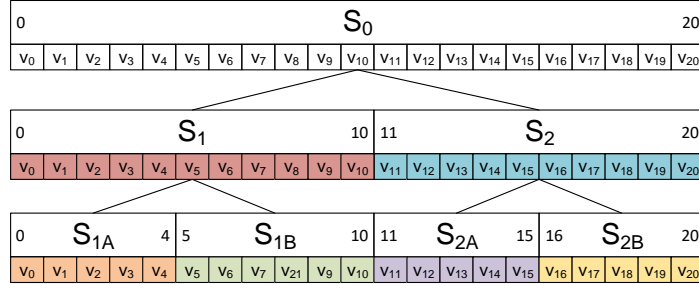

Figure 3: Hierarchical Subgraph Vertex Ordering

Improved Dijkstra Implementation. Array-based data structures offer significantly better practical performance due to their locality of reference compared to hash-tables, enabling speedup through efficient utilization of CPU cache [25]. As another benefit of replacing hash-tables, the subgraph vertex ordering allows substantial improvement in the implementation efficiency of SUL-Tree construction. Specifically, SDL computation involves a restricted Dijkstra search which terminates when all subgraph vertices are settled. Checking whether a settled vertex is a subgraph vertex can be performed by simply checking if the vertex ID is in the range for that subgraph. Moreover, the distance itself can be stored directly in an array-based SDL. Construction time can be reduced by 33% from simply avoiding hash-tables (Table 7).

8.2. Subgraph Restricted Dijkstra Search

The most intensive part of SUL-Tree construction is the creation of SDLs for each SUL-Tree node n_T. This involves a Dijkstra search from each of the m landmarks to find the shortest path distances to all vertices contained in the subgraph associated with n_T. Although the recursive partitioning step ensures subgraphs are disjoint vertex sets, there is still potential for repeated work as there is no guarantee that

shortest paths will *only* involve subgraphs vertices. It is possible that a shortest path from a landmark to a subgraph vertex leaves and re-enters the subgraph. To ensure correct distances, the Dijkstra search cannot terminate until all subgraph vertices are settled. In Section 7, we used γ to represent the average factor by which subgraph Dijkstra search exceeded its number of vertices. Ideally, $\gamma = 1$, meaning only subgraph vertices were settled. While it is small in practice ($\gamma \approx 6.6$), it can be further reduced using a novel variant of Dijkstra inspired by COL-Tree search, as we describe next.
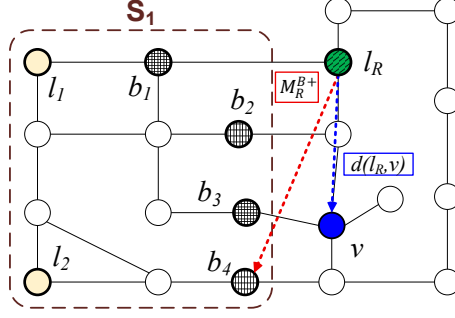
Figure 4: Lower-Bound Distance to Subgraph Borders

Border Set Lower-Bounds: We observe that the COL-Tree object search mechanism can be used to improve Dijkstra subgraph search as well. First, we ensure that SUL-Tree nodes are processed in partitioning order, so that root node SDLs can be accessed during SDL computation for any descendant node. After a subgraph is generated from the partitioning step, we identify the border vertex set B, which contains the subgraph vertices that are adjacent to a non-subgraph (i.e., external) vertex, e.g., $B = \{b_1, b_2, b_3, b_4\}$ in Figure 4. We also keep track of the max M_R^{B+} and min M_R^{B-} distance from each root landmark l_R to any border vertex by retrieving landmark-border distances from the SDL of the SUL-Tree root node.

The optimization uses the fact that any shortest path between subgraph vertices through an external vertex must re-enter the subgraph through some border vertex. Thus, we can compute a lower-bound from an external vertex v to the subgraph by using M_R^{B+} and M_R^{B-} for root landmark l_R using Eq. (4) and choose the best lower-bound by Eq. (5). This is similar to the way COL-Trees are used to compute lower-bounds to objects contained in a subgraph (in this case, the borders are essentially the objects). $d(l_R, v)$ for use in Eq. (4) is already available and can be retrieved in $O(1)$ time from the root SDL_R for l_R. For example, in Figure 4, the shortest path from subgraph landmark l_2 to some subgraph vertices may leave through b_4, pass through vertex v, and re-enter via b_3. A lower-bound from v to any border of subgraph S_1 can be computed using already available $d(l_R, v)$ and M_R^{B+} for root landmark l_R, potentially pruning the Dijkstra search at v. Note that the heuristic may be inconsistent, as vertex v further from source l may have a smaller lower-bound $d(l, v) + LB(v, B)$ to any subgraph vertex. While still admissible (as a lower-bound on the true distance), we employ the widely used pathmax heuristic [32] to ensure the lower-bound to the subgraph $d(l, v) + LB(v, B)$ does not decrease, allowing use of faster priority queues such as radix heaps. This is still correct because $d(l, v) + LB(v, B) \geq d(l, u) + LB(u, B)$ must be true if the shortest path to v goes via u, i.e., there cannot be a strictly shorter path back to the subgraph via u by moving away from it.

We can similarly compute an upper-bound from the source l to any border vertex using Eq. (5) and prune the search at external vertex v if $d(l, v) + LB(v, B) > UB(l, B)$. However, this upper-bound may be loose if all root landmarks are far from the subgraph, therefore we supplement it by maintaining an upper-bound on the distance between any two subgraph vertices while computing SDLs (similar to graph diameter, it is $2\times$ the maximum distance from subgraph source l to any node).

Object Set	Size	Density
Schools	160,525	0.007
Parks	69,338	0.003
Fast Food	25,069	0.001
Post Offices	21,319	0.0009
Hospitals	11,417	0.0005
Hotels	8,742	0.0004
Universities	3,954	0.0002
Courthouses	2,161	0.00009

Table 3: Real-World POIs Sets for US Road Network

Variable	Values
k	1, 5, **10**, 25, 50
d	1, 0.1, 0.01, **0.001**, 0.0001
A (%)	1, 5, **15**, 50, 100
$|Q|$	2, 4, **8**, 16, 32
Agg. Func.	*sum*, ***max***
r (% of D)	0.5, 1, **2.5**, 5, 10

Table 4: Experimental Variables (defaults in **bold**)

9. Experiments

9.1. Experimental Settings

Environment: We conduct experiments on a PC with a AMD Ryzen 7 9700X CPU with 64GB of RAM running Ubuntu 24.04 LTS (64-bit). Out of an abundance of caution, we disable CPU features such as SMT and PBO to avoid any possible inconsistencies, e.g., due to unexpected changes in clock-speed. Code was written in C++ and compiled by g++ v13.3.0 with O3 flag and is publicly available as a Bitbucket repository[2]. All experiments were conducted using memory-resident indexes for fast query processing.

Datasets: We evaluated methods on a real large-scale road network for the continental US with $23,947,347$ vertices and $57,708,624$ travel time edges combined with sets of locations for 8 types of POIs as listed in Table 3. These are curated versions of a road network obtained from the 9th DIMACS Challenge[3] and POI locations obtained from OSM[4], and have been made available via Zenodo [33]. For sensitivity analysis, we generate synthetic object (POI) sets from vertices chosen uniformly at random for density $d=\frac{|P|}{|V|}$.

Experimental Variables: Table 4 lists all variables evaluated (bold indicates the default value). For AkNN and kFN queries, we perform sensitivity analysis on query performance by varying the number of results k and object set density d. For AkNN queries we additionally evaluate varying number of query vertices $|Q|$, aggregate functions, and variable A representing how "local" the set of query locations are to each other. Similar to past studies [4], A refers to a connected subgraph of the road network G with $A\%$ of the total vertices $|V|$, from which $|Q|$ random query vertices are chosen. Note that we set a low default of $A=15\%$ to ensure fairness, as our method is likely to perform better when query locations are further apart with higher A. Query vertices are chosen uniformly at random for other queries as in previous studies [25, 3]. For range queries, in addition to d, we evaluate radius r expressed as a percentage of the graph diameter D of the road network (which we approximate by the double-sweep method).

Performance Metrics: We report average running time of 1000 sequentially executed queries. For sensitivity analysis, this consists of 20 object sets and 50 query vertices (kFN and range) or vertex sets (AkNN). In addition to query time, we measure heuristic efficiency using two machine-independent metrics, the total numbers of (a) network distances computed to objects; and (b) candidate objects retrieved from the data structure. The former measures how accurate the lower-bounds are at avoiding expensive network distance computations to objects that are not results and the latter measures how effective each data structure is at implicitly pruning non-result objects (i.e., by never even considering them individually).

Competitors: We compare the performance of each query algorithm that utilizes COL-Trees against state-of-the-art heuristics for POI search including the hierarchical search-based *Incremental Euclidean Restriction* (IER) [2] and expansion-based *Network Voronoi Diagrams* (NVD). IER has been applied to solve AkNN queries [4]. We significantly improve existing NVD-based AkNN search [16] by adapting Incremental Lower-Bound Restriction [18] that uses ALT to avoid expensive network distance computation. Note that IER
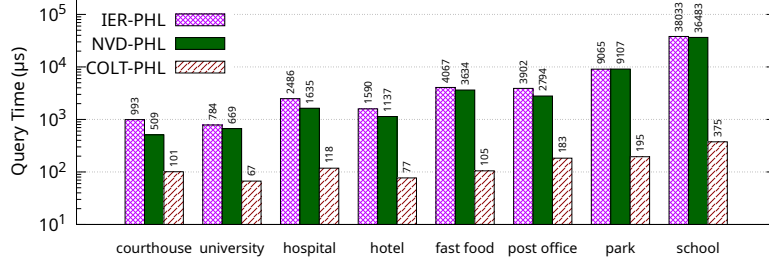

Figure 5: AkNN query performance on different real-world POI sets

cannot solve kFN queries and NVD will always evaluate all objects (see Section 5), so we implemented an optimized brute-force baseline, AUB-PHL, that evaluates all objects but filters candidates by ALT upper-bounds to reduce network distance computation. Comparisons for each query are listed in Table 5. Note that while COL-Trees are not designed with kNN queries in mind, we provide experimental results in Appendix A as there are some cases it can be highly useful (e.g., where NVD indexing cost cannot be amortized).

Technique	AkNN	kFN	Range	kNN
COLT-PHL (Ours)	✓	✓	✓	✓
IER-PHL [4]	✓	✗	✓	✓
NVD-PHL [18]	✓	✗	✓	✓
AUB-PHL	✗	✓	✗	✗

Table 5: Compared Methods for Each Query

Each technique uses Pruned Highway Labeling (PHL) [27] for exact network distance computation, hence techniques are named IER-PHL, NVD-PHL, COLT-PHL, etc. Using the same network distance technique ensures a level playing field. Furthermore, as PHL is one of the fastest techniques and network distance is the most costly subroutine, it minimizes the bottleneck and better exposes overheads associated with each technique. NVD-based methods use the ALT index [20] with m=16 random landmarks to compute LLBs and NVDs are represented as linear quadtrees [34, 35] for reduced space cost and improved memory locality. We also utilize the same ALT index in query algorithms based on COL-Tree to compute point-to-point lower- and upper-bounds, as it is effectively a SUL-Tree root node, for simplicity and a means of like-for-like comparison. We select COL-Tree parameters for each query by parameter testing detailed in Section 9.6.

9.2. AkNN Queries

Figure 5 depicts AkNN query time on the real-world POI datasets, with the number of objects increasing from left to right. COLT-PHL significantly outperforms the other methods across the board, with up to 2 orders of magnitude improvement. COLT-PHL tends to improve more on larger POI sets, where it is more difficult to distinguish between objects. Next, we perform sensitivity analysis to delve deeper into this and other nuances of query performance.

9.2.1. Sensitivity Analysis for AkNN Queries

Effect of d: The trend seen for real-world POIs is confirmed by increasing object density d in Figure 6a. Both IER and NVD scale poorly with increasing d. With more objects, NVD-PHL must expand more adjacent objects to find a common candidate for the same query vertices, while IER-PHL finds it harder to distinguish objects using its less accurate Euclidean lower-bound. In contrast, COLT-PHL benefits from the tighter lower-bounds and hierarchical traversal afforded by the COL-Tree data structure, resulting in effective pruning of subgraphs and pinpointing likely candidates.
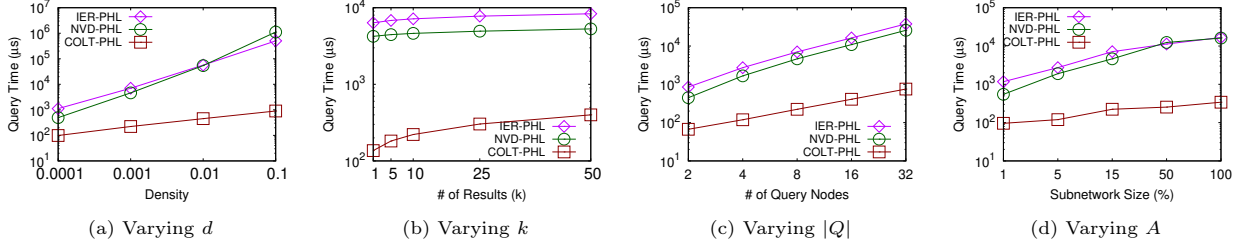
Figure 6: Performance for *max* function

Effect of k: COLT-PHL significantly outperforms the other methods for varying k in Figure 6b. NVD-PHL and IER-PHL query times do not vary significantly compared to COLT-PHL (note the logarithmic scale). This suggests the same amount of work is done irrespective of k and strongly implies the competing techniques cannot effectively identify good candidates and terminate quickly. For example, the fact that subsequent result objects are not much more costly to retrieve suggests the concurrent expansion in NVD-PHL is extremely large just to generate the first candidates.

Effect of $|Q|$: As the number of query vertices increases in Figure 6c, additional lower-bounds and network distances to candidates will be computed, thus query time increases for all methods. However, COLT-PHL scales better due to the convexity-preserving property only requiring a single binary search on the COL-Tree's ODLs irrespective of the number of query vertices. On the other hand, NVDs require additional concurrent expansions as AkNN results are less likely to be close to any single query vertex.

Effect of A: Recall that A is the percentage of graph vertices in a subgraph from which query vertices Q are chosen. With increasing A query vertices become further apart, e.g., to represent a query by a logistics company placing depots nation-wide. For $A = 1\%$, NVD-PHL is closer to COLT-PHL in Figure 6d. This scenario resembles kNN queries where NVD-PHL is effective, as more query vertices share the same 1NN and concurrent expansions overlap earlier. IER does not benefit from this as its lower-bounds are still inaccurate. Queries become harder with increasing A and COLT-PHL scales extremely well in that case.
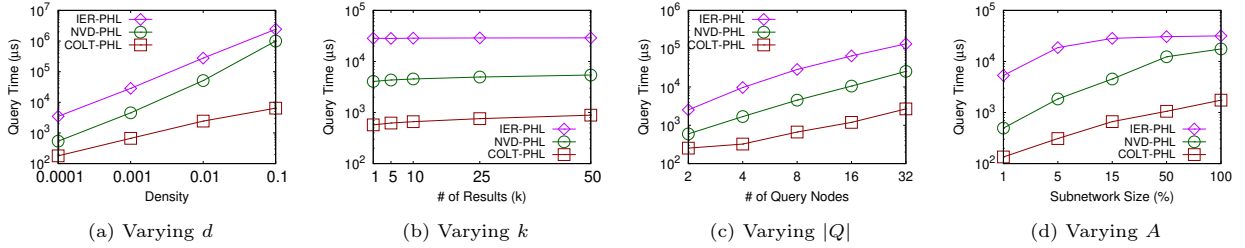

Figure 7: Performance for *sum* function

Effect of Aggregate Function: We evaluate another popular aggregate function, *sum*, in Figure 7. Both IER-PHL and COLT-PHL have higher query times for *sum* than *max* because *sum* also accumulates the error of the lower-bound. This effect is amplified by the hierarchical tree search used by both methods, explaining the relative improvement of NVD-PHL. However, tighter lower-bounds in the COL-Tree hierarchy make COLT-PHL more robust to this than IER. NVD-PHL efficiency is dominated by the concurrent expansions that must meet to generate candidates, which remains similar for *sum*. Regardless, COLT-PHL still significantly outperforms both methods.

Heuristic Efficiency: Figure 8 and 9 shows the heuristic performance for *max* and *sum*, respectively. The significantly higher number of network distances computed to objects in Figures 8a and 9a explain the poor query time of IER. However, NVD-PHL does not compute many more network distances than COLT-PHL. As both methods use landmark lower-bounds (LLBs), that are more accurate than IER's Euclidean-based lower-bound, they both avoid computing network distances. The poor query time of NVD-PHL can be

explained by Figures 8b and 9b, which shows that NVD-PHL considers significantly more candidate objects. This is because the concurrent expansions from each query vertex must continue to retrieve candidates until they meet before even trying to compute network distances, which is not efficient for AkNN search. Note that we exclude IER from Figure 8b (and other figures for candidates retrieved) because the metric is designed to differentiate the landmark-based techniques, NVD-PHL and COLT-PHL.
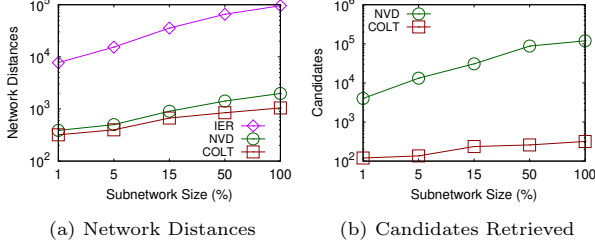

(a) Network Distances	(b) Candidates Retrieved

Figure 8: Heuristic Performance (max)

(a) Network Distances	(b) Candidates Retrieved
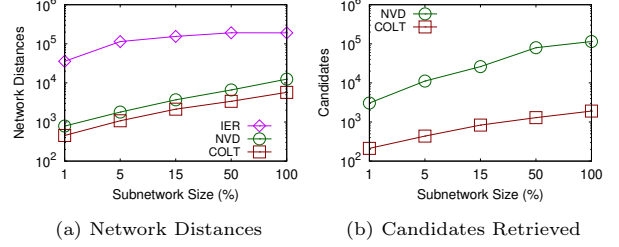
Figure 9: Heuristic Performance (sum)

9.3. kFN Queries

The effectiveness of COLT-PHL for kFN queries on real-world POI datasets is shown in Figure 10. Recall that IER-PHL cannot be used to solve kFN at all and concurrent expansion using NVDs can be no better than evaluating all objects, thus we can only compare to an optimized brute-force baseline, AUB-PHL. Consequently, the COLT-PHL maximum speedup is even greater than AkNNs, being up to 330× faster.

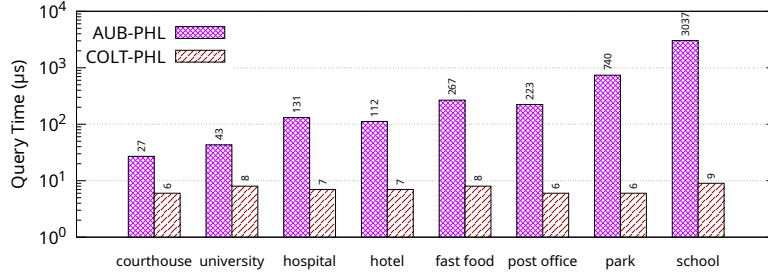Figure 10: kFN query performance on different real-world POI sets

9.3.1. Sensitivity Analysis for kFN Queries

Effect of d: COLT-PHL remains largely unaffected by varying density d in Figure 11a. This indicates that the pruning ability is highly effective. With increasing density, the resulting COL-Tree will simply have a deeper hierarchy and effective bounds will prune the additional branches and thereby levels. Since AUB-PHL must evaluate all objects, it is only competitive when object sets are extremely small.

Effect of k: Given that it must inspect all objects, AUB-PHL must essentially do the same amount of work irrespective of the number of results needed, as evident with increasing k in Figure 11b. On the other hand, COLT-PHL scales more reasonably with increasing k. One noteworthy observation is the comparison between query times for kFN and its opposite, the kNN query. kFN query times for COLT-PHL are faster than kNN (Figure A.19b) despite using the same data structure in much the same way. The difference is that kFN traversal is by upper-upper instead of lower-bound, suggesting the former is more effective.

Heuristic Efficiency: The extremely low number of network distances in Figure 11c suggests that COL-Tree upper-bounds are highly accurate, and mostly retrieve candidates that are one of the kFN results. Upper-bounds are known to be much closer to the true network distance [30] and being able to prune based on them shows that COL-Trees are highly suited for kFN queries. The uptick for higher densities is due to

objects being far closer together, which reduces the margin of error, affecting even accurate upper-bounds. This is confirmed in Figure 11d, because the number of candidates does not similarly increase.
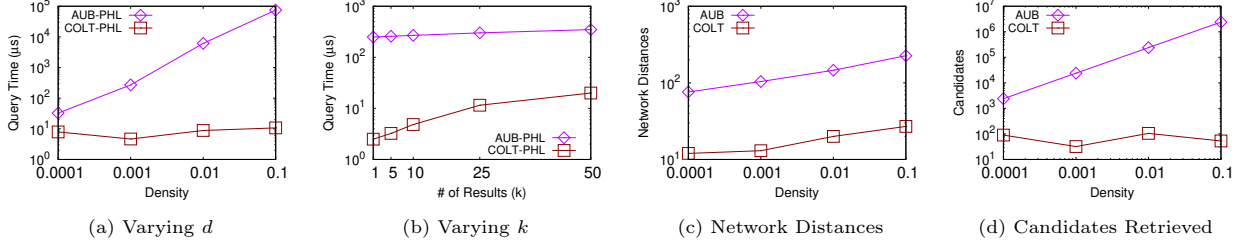

(a) Varying d (b) Varying k (c) Network Distances   (d) Candidates Retrieved

Figure 11: Performance for kFN Queries

9.4. Range Queries

COLT-PHL range query times are surprisingly on par with NVD-PHL with the smallest object sets in Figure 12, gradually improving with increasing size. On the largest set it achieves a significant $2.7\times$ improvement. Recall that the motivation for COL-Trees was based on POI search queries such as AkNN and kFN that are not aligned with the "nearest first" heuristics more suited for kNN and range queries. Intuitively, COLT-PHL should not work so well for range queries and we investigate why next.
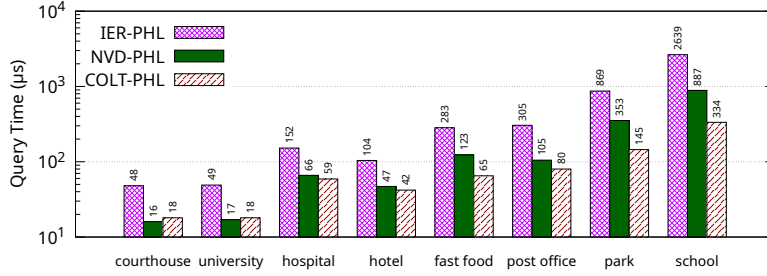

Figure 12: Range query performance on different real-world POI sets

9.4.1. Sensitivity Analysis for Range Queries

For range search sensitivity analysis, we increase the default density to d=0.01 to make queries more challenging and retrieve more objects as results (our varying d experiment still evaluates all densities).

Effect of d: The query time of COLT-PHL improves relative to NVD-PHL with greater density d in Figure 13a. With increasing density, more objects fall into the same range, and COLT-PHL scales better as more objects can simply be identified as results or pruned using only the COL-Tree node lower- and upper-bounds to entire groups of objects (i.e., without retrieving individual objects from leaf node ODLs). COLT-PHL achieves significant improvement on higher densities. We must also take into account that building an NVD index is more than $1500\times$ slower than a COL-Tree (Table 6), making methods based on COL-Trees clearly preferable.

Effect of r: COLT-PHL also improves relative to NVD-PHL with increasing radius r in Figure 13b. Similar to the case with increasing d, increasing r leads to more objects being evaluated and returned as results, which COL-Tree traversal can handle well. The scalability of COLT-PHL also likely benefits from two advantages with increasing r. First, range search within COL-Trees can be implemented by a stack while NVD-PHL must use a priority queue (such as a binary heap). Intuitively, similar to increasing a circle's radius, a modest increase in range can dramatically increase the area covered and the potential number of objects within range. While not directly comparable, stacks will scale better for the corresponding significant

increase in data structure operations than priority queues. Second, as already discussed in relation to kFN queries, we know that upper-bound pruning is extremely effective and increasing r means more objects fall within it and can be immediately confirmed as results by upper-bounds to entire groups of objects.

Heuristic Efficiency: Range queries do not strictly require computation of network distance to each object, so as long as we correctly determine they are within range r. Figure 13c shows that COLT-PHL is able to compute fewer network distances to objects with increasing d, suggesting that its upper-bounds are indeed working as more objects fall within the same range, as described above. Moreover, COLT-PHL is more agreeable with increasing d compared to queries that retrieve k results, since the order is not relevant for range queries. Figure 13d shows that COLT-PHL scales better with increasing d on the number of candidates considered (i.e., objects that can neither be implicitly pruned or immediately reported as results), eventually overtaking NVD-PHL. Such behavior would penalize NVD-PHL's priority queue more severely.
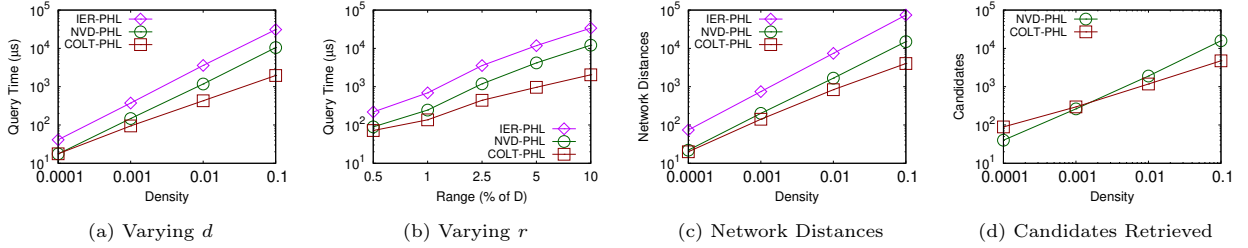

Figure 13: Performance for Range Queries

9.5. Pre-Processing Performance

Table 6 details pre-processing costs in terms of time and space. Naturally, indexes for the road network are more costly. SUL-Tree indexing time only represents a small overhead over the other landmark-based index, ALT, aligning with our theoretical analysis in Section 7. Index size is even lower, as fewer (but more localized) landmarks are need to support search, offsetting the additional $O(\log |V|)$ compared to ALT. Moreover, the indexing time of SUL-Tree is still extremely low and significantly lower than PHL, which possesses one of the fastest pre-processing times for high-performance road network indexes [27]. ALT is constructed for m=16 landmarks and SUL-Tree is based on m=2, b=8, and α=1024.

Object index (COL-Tree, NVD, R-tree) statistics are reported for the default density $0.001 \times |V|$. While these costs seem trivial it is still important as there may be many object sets for different types of POIs like those in Table 3. Moreover, the pre-processing time determines how easily a technique can be used in real-time settings. For example, in a ride-hailing application, the objects are moving and object indexes must be rebuilt frequently [36]. COL-Tree is significantly smaller and faster to construct than NVDs and is comparable to R-trees as both possess the same asymptotic behavior. Statistics are reported for an R-tree with branching factor 32 and COL-Tree with m=2, b=8, and λ=256.

Note that for SUL-Trees, in theory, we require $\alpha=\lambda$ in case all vertices in a SUL-Tree leaf node are objects. However, in practice, we find that we can safely set $\alpha=4\times\lambda$ without affecting COL-Tree construction at all, even for the largest object sets (d=0.1). This reduces SUL-Tree construction time by avoiding computing $SDLs$ that are never used. If a case where smaller α is needed occurs, we can simply handle it on the fly via SDL computation on the corresponding (unprocessed) SUL-Tree node.

9.5.1. Pre-processing Improvements

We evaluate the SUL-Tree pre-processing improvements proposed in Section 8 with results detailed in Table 7. The first row reports the impact of the subgraph vertex ordering (Section 8.1) on index size. By eliminating the need to store vertex IDs anywhere in the SUL-Tree, this significantly reduces the index size by almost 50%. The second row shows 33.3% improvement of SDL construction time by using the new ordering to eliminate the use of hash-tables during the subgraph Dijkstra search.

Index	Time	Space
SUL-Tree	*2m19s*	*1.15GB*
ALT	39s	1.46GB
PHL	14m17s	16.2GB
COL-Tree	*3.4ms*	*0.5MB*
R-tree	2.7ms	0.9MB
NVD	5392ms	28MB

Table 6: Road network and object index preprocessing costs (based on US road network and uniform object sets with d=0.001)

Optimization	Before	After	% Improv.
Subgraph Ordering	2263MB	1146MB	49.4%
No Hashtable Dijk.	129s	86s	33.3%
Border Set Dijk.	86s (γ=6.6)	65s (γ=4.7)	24.4%

Table 7: Impact of SUL-Tree pre-processing optimizations described in Section 8 (SUL-Tree parameters used: m=2, b=8, and α=1024)

The final row reports the impact of the restricted Dijkstra subgraph search using border set lower-bounds (Section 8.2) in terms of improved runtime and reduction in γ. Recall that γ is the average factor by which subgraph Dijkstra search expansion exceeds the number of subgraph vertices. The SDL construction time after the removal of hash-tables is reduced by a further 24.4% using the new variant of Dijkstra search. This is slightly lower than the percentage improvement of γ (28.8%). While the γ improvement shows a smaller proportion of vertices are visited than the runtime improvement, the difference is explained by the small overhead added by computing lower-bounds. Note that the reported times are specifically for subgraph Dijkstra search (i.e., populating SDLs) and excludes other tasks for SUL-Tree construction, such as partitioning, which are not targeted by any of our proposed optimizations.

9.6. Index Parameter Testing

We investigate query performance while varying the structure and properties of COL-Tree on four variables: the branching factor b, the maximum leaf node ODL size λ, the number of landmarks per node m, and the landmark selection policy. The tested values for each are listed in Table 8. While COL-Tree outperforms competing methods even with naive values for parameters (e.g., m=2 random landmarks), valuable performance improvements can be extracted by carefully choosing them.

A Note on Parallelization: While SUL-Tree construction is fast (taking no more than a few minutes), we use parallelization to make exhaustive parameter testing on hundreds of SUL-Trees feasible in only a few hours. The SUL-Tree node SDL computation was easily parallelized using OpenMP, but we leave the full exploration of parallelization to future work. Note that we only use multi-cores during parameter testing for SUL-Tree construction (any reported time is single-threaded). We next present query performance of COLT-PHL with tested parameter values listed in Table 8, along with finalized parameters for each query.

Variable	Tested Values	AkNN	kFN	Range
b	4, 8, 16, 32	8	4	8
λ	256, 512, 1024, 2048	256	256	256
m	2, 4, 8, 16	2	4	2
Landmark Type	Random, Minmax, Farthest, Slice	Random	Slice	Minmax

Table 8: Tested values and finalized COL-Tree parameters on US dataset for each query

Figures 14 to 17 illustrate query time with varying COL-Tree parameters for each query type. While the parameter indicated in each figure is varied the remaining parameters take on the default values for that query indicated in Table 8 (see Table 4 for other defaults). The overarching observation is that in most cases varying parameters does not significantly affect COL-Tree query performance, indicating that performance is not beholden to extensive parameter tuning. We analyze the noteworthy cases below:

- For AkNN queries, query performance degrades with increasing m in Figure 16a. Recall that we must choose the tightest lower-bound over all landmarks as per Eq. (2). Hence, with increasing m, the cost of computing lower-bounds increases, which must be compensated by increased lower-bound accuracy.

This is a less viable trade-off for AkNN queries, because even if we have more landmarks, it is still difficult to find one that is suitable for all query vertices. Thus, we simply take longer to compute similar quality lower-bounds with increasing m, making $m=2$ the best choice.

- kFN query performance is generally poorer for larger b (Figure 14b) and λ (Figure 15b). The query performance also fluctuates. This is likely caused by large values for b and λ being the most likely scenarios where extremely skewed tree structures can form. For example, if λ is too large it could result in one ODL with close to λ objects while another with significantly less. Moreover, given the extremely low query times for kFN queries, it is more sensitive to minute changes in performance.

- Range query performance with different landmark types shows a preference for the Border Minmax method in Figure 17c, with the improvement widening as density d increases. These landmarks attempt to minimize the distances to all vertices in the subgraph. As a result they are more likely to produce the smallest minimum and maximum distances to objects on average. This is beneficial to range queries, as a small range of landmark-object distances increases the likelihood of being able to report whole swathes of objects immediately as results or prune them entirely.

Effect of Landmark Selection Policy: While the above discussion was concerned with noteworthy differences between certain parameters, query performance for varying landmark selection methods is noteworthy due to lack of significant difference in Figure 17. *Border Minmax* is marginally better on higher densities for range queries, while *Slice-Based Furthest Border* mostly performs better for kFN queries. The latter makes intuitive sense for single query vertex queries like kFN queries, which benefit from improved upper-bounds when landmarks are "between" the query vertex and objects. Choosing borders from Euclidean slices ensures all angles of entry are covered, which is a common concept in road network shortest path computation, e.g., *transit nodes* [37]. On the other hand, AkNN queries benefit from landmarks being "behind" multiple entry points due to multiple query vertices, meaning centrally located landmarks chosen by Border Minmax are more effective. But the intensive selection process makes random vertices more attractive for AkNN, as the difference is not significant. On one hand, the results suggests COL-Trees will perform reasonably well regardless of the landmarks selected. Nonetheless, it also implies landmark choice warrants further study.
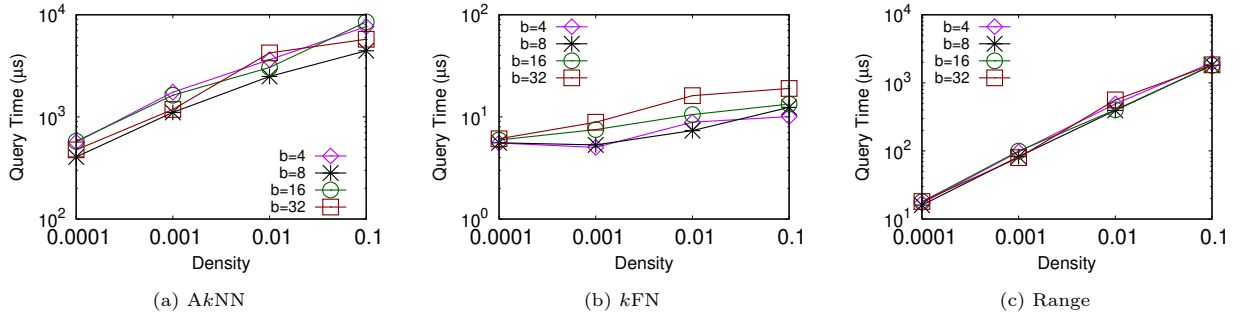

(a) AkNN   (b) kFN   (c) Range

Figure 14: Query Performance with Varying Branching Factor b

10. Related Work

In this section, we review additional literature related to object search queries, road network indexing techniques, and other work broadly related to landmarks.

Aggregate Nearest Neighbors: In Section 1, we extensively discussed the shortcomings of the state-of-the-art AkNN techniques for road networks [4, 16], which motivated COL-Tree. The IER-based technique suffers from the inaccuracy of Euclidean distance, especially when weights represent the more realistic travel time along edges. This is a more serious problem in AkNN search as the error will also be aggregated, further limiting search space pruning. The NVD-based concurrent expansion suffers from high
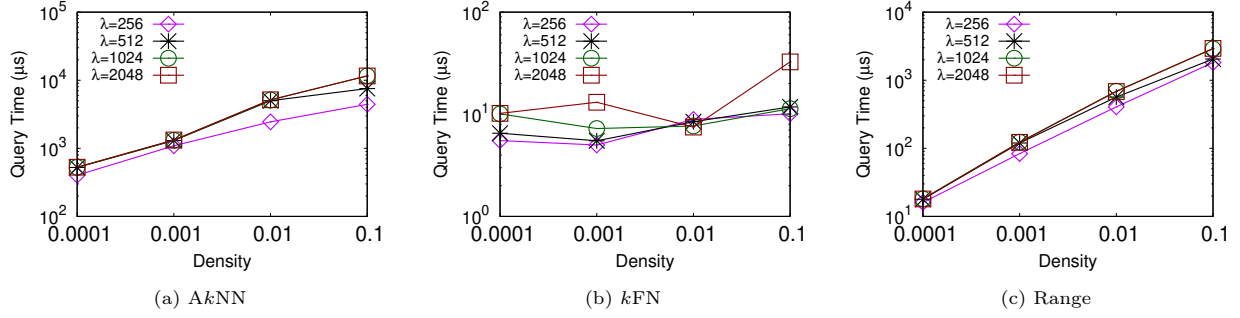
(a) AkNN

(b) kFN

(c) Range

Figure 15: Query Performance with Varying Object List Size λ

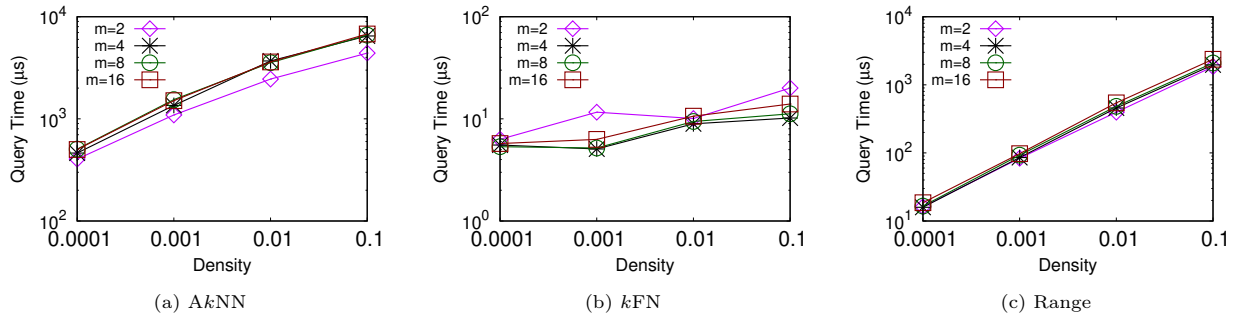
(a) AkNN

(b) kFN

(c) Range

Figure 16: Query Performance with Varying Number of Landmarks m

pre-processing costs and excessive expansions before candidates can be generated. All previous variants of AkNN search such as Flexible ANNs [17, 38] or involve moving objects [39] are derived from these less desirable heuristics. One recent work on Flexible ANNs [40] (published after the conference version of our work) utilizes M-trees [41], which resembles a simplified version of COL-Tree that does not utilize graph partitioning, tightening lower-bounds over multiple landmarks (i.e., references nodes), and other features of COL-Trees such as subgraph vertex ordering that make it highly suited to road network graph search.

Farthest Neighbors: Farthest neighbor search has not been widely studied in the context of road networks. Bose et al. [11] propose a theoretical method for Farthest Point Diagrams in planar graphs, a dual to NVDs. NVD construction involves an expensive Dijkstra search from each object, but only needs to settle a vertex once as each search can prune vertices already settled by another search. FPDs cannot benefit from this, potentially settling $|V|^2$ vertices for *each object set*, which is significantly more expensive than even road network index construction. As a result, this approach is not practicable in most real-world applications. Other works [42, 39, 43] incorporate the farthest metric into variants of kNN queries but utilize typical kNN heuristics. COL-Tree, which enables efficient incremental upper-bound-based candidate retrieval, would be highly suited to better solving these farthest query variants.

k **Nearest Neighbors:** Early kNN methods involved creating a single index incorporating the object set P and the road network G together, which is then used to both find objects and compute distances. For example, Network Voronoi Diagrams were first utilized in VN^3 [44], where the index includes distances between Voronoi regions to simultaneously compute distances to new candidates. Furthermore, many of these early techniques only supported fixed k [45]. Most recent techniques have proposed road network indexes that are utilized by multiple *object indexes*, e.g., G-tree and its Occurrence Lists [3]. Object indexes are typically much less costly to compute and, as previously observed [25], this *decoupling* is highly advantageous and aligns more clearly with real-world usage. For example, a ride-hailing service frequently computes kNNs as drivers move and new passengers make requests. Thus, the object indexes change much more frequently than the road network, benefiting from fast rebuilding. Moreover, this decoupling has allowed more in-depth study into the candidate retrieval heuristics. For example, Abeywickrama et al. [25] found that a
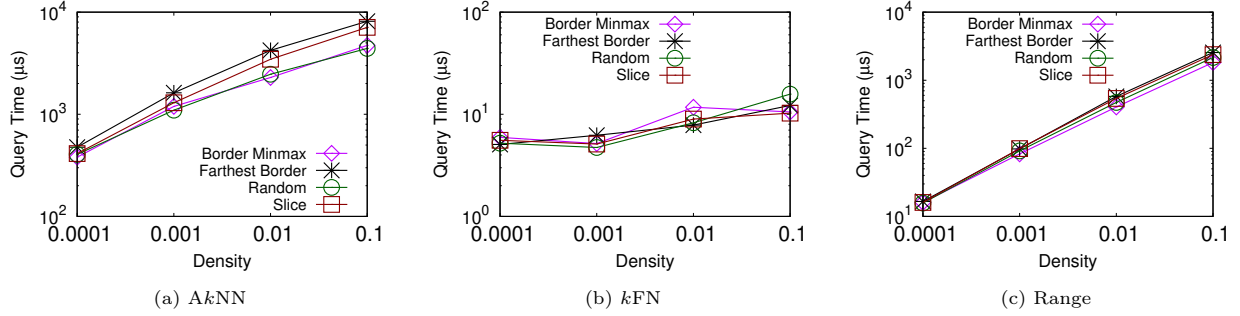
25

Figure 17: Query Performance with Varying Landmark Type

Euclidean distance heuristic could more efficiently utilize the G-tree index than its own search heuristic. It is worthwhile to underscore these benefits, as recent work [46] has reverted to non-decoupled indexes and fixed k. We refer the reader to a recent experimental review [25] for a detailed overview of kNN techniques.

Road Network Indexes: Indexing techniques to more efficiently answer network distance queries on road networks have been extensively studied for several decades [26]. Advancement in this area is orthogonal to our work, as object search utilizing COL-Tree is decoupled from the network distance computation. Thus, faster network distance querying will further improve the performance of COL-Tree. Typically, the lowest query times are provided by the most pre-processing intensive indexes, e.g., 2-hop labels. We chose PHL [27] because it provides one of the best trade-offs between network distance query time and pre-processing time for 2-hop label techniques. Nonetheless, COL-Tree can easily utilize the latest 2-hop labeling methods [47] with faster queries but more expensive pre-processing. The powerful flexibility offered by a decoupled index allows the real-world system designer to choose a method that matches the amount of pre-processing they can tolerate. For example, as an alternative, if the road network index must be rebuilt or updated frequently, the user can choose a lower cost index such as G-tree [3] or Contraction Hierarchies [48].

Landmarks and Other Search Heuristics: In road networks, landmarks were originally utilized as an improved heuristic for A* shortest path search [20]. This extensively studied concept is also referred to as differential heuristics [49] or beacons [50]. Many related works focus on reducing the storage and memory costs, e.g., computing landmark distances to only part of the graph and compensating for this by computing distances between the landmarks [51]. Other work [52] has focused on improving accuracy, e.g., through better landmark selection by combining landmarks with *reach*-based pruning. Many of these works are complementary to our techniques, e.g., integrating reach into COL-Tree could improve heuristic pruning and reduce false candidates or Compressed Differential Heuristics [23] could improve lower-bound computation time without loss of accuracy. Other notable heuristics include Euclidean distance-based heuristics [53], compressed path databases [54], portal-based true-distance [55], and FastMap [56] heuristics.

Landmark-Based Object Search in Road Networks: Past attempts [51] to leverage road network LLBs for object search based on the multi-step kNN paradigm [57] required a ranking that necessarily involved computing lower-bounds to all objects. As discussed, this approach is not scalable with object set size and does not compare favorably to methods that are capable of heuristic pruning. This is confirmed by the performance of the baseline kFN method, which takes a similar approach. Road Network Embedding (RNE) [58] involves transforming the road network into higher-dimensional space and using Minkowski metrics to estimate network distance. However, the proposed kNN method is approximate. Qiao et al. [59] propose to use shortest path trees to compute distance estimates based on tree distance, but their technique is also approximate [59] and is applied to the alternative *keyword* search problem in road networks. In contrast, COL-Tree provides a general data structure with an admissible heuristic that can be employed to answer a variety of road network queries with exact results.

Landmark-Based Search in Other Contexts: Mouratidis et al. [60] proposed computing landmark lower-bounds to groups of users in a social network similar to Eq. (4). However, their technique is not designed for hierarchical graph traversals and, unlike COL-Tree, does not address the problem of computing

26

more accurate lower-bounds (e.g., with more landmarks) necessary for an effective hierarchical search. M-trees [41], typically used for similarity search and multimedia databases, can be considered as a simplified version of COL-Tree. However, applying them to hierarchical graph traversal is not straightforward, with road networks in particular presenting domain-specific challenges. For example, COL-Trees utilize a SUL-Tree to significantly reduce construction time through pre-processing the road network first and use multiple landmarks (i.e., references nodes) to obtain better bounds. Landmark upper-bounds have been studied as a means for distance estimation in large-scale networks with the observation that they tend to be closer to the true network distances than lower-bounds [30], but the utility in search was not considered.

11. Conclusion & Future Work

We present COL-Tree, an efficient index to support hierarchical search in road network graphs using landmark lower-bounds. We show that COL-Trees are highly suited to several queries where traditional search heuristics fall short: AkNN, kFN, and range queries. For example, AkNN query results are often not close to any single agent location and are more easily located using a hierarchical subgraph traversal supported by COL-Tree. Combined with its novel property for convexity-preserving aggregate functions, we can retrieve more promising AkNN candidates and terminate the search sooner. Moreover, the proposed data structure is light-weight in both theory and practice. Lastly, our experiments revealed significantly improved performance of up to several orders of magnitude, particularly for AkNN and kFN queries.

Future Directions: COL-Tree is a versatile data structure with potential application to other heuristic search problems. For example, we believe COL-Tree has the potential to be widely used in a similar way to R-trees in location-based services (LBS). While road network distance is more accurate than Euclidean distance, many LBS rely on R-trees for lower pre-processing cost despite poor accuracy. As COL-Tree has significantly reduced the gap, it is now more practicable to consider revisiting graph-based approaches for challenging problems such as Reverse kNNs [61]. In fact, COL-Tree has recently been applied to improve real-world driver-passenger matching in ride-hailing [36]. There are also many possible directions to improve heuristic performance, such as landmark selection or applying Compressed Differential Heuristics [23].

Appendix A. kNN Querying with COL-Tree

Recall one of the motivations for COL-Tree was the fact that most existing heuristics were designed with kNN queries in mind and did not translate well to other problems such as AkNN and kFN search. However, in Section 9.4, we found that COL-Tree could still perform on par or better than state-of-the-art kNN heuristics for range queries due to certain advantages offered by COL-Tree. This was particularly beneficial given the significantly lower pre-processing cost of COL-Trees. In this appendix we investigate the performance of COL-Tree for kNN queries to illustrate scenarios where it could be a better option. We omit the pseudocode for the kNN query algorithm, as it can be easily derived by simplifying Algorithm 1 based on the assumption that the query vertex set Q contains only one vertex.

Appendix A.1. Experimental Results

As expected, the state-of-the-art NVD heuristic offers the best performance on real-world POI datasets in Figure A.18, with COLT-PHL coming second in all cases. The performance gap between NVD-PHL and COLT-PHL is negligible on small POI sets because the heuristics need only differentiate between a small pool of candidates to determine exact results. Nonetheless, even on the largest datasets, the difference is not significant. For example, the worst performance for COLT-PHL is for schools where it is 2.5× slower, but the difference is only 21μs. Based on Table 6, a system using NVDs would take 5.55s to construct an index

and process a workload of 10,000 kNN queries. In that time, a COL-Tree could be constructed and serve more than 150,000 queries, making it preferable when amortization opportunities are limited. COL-Tree parameters for kNN querying were $m=4$ slice-type landmarks, $b=8$, and $\lambda=256$.
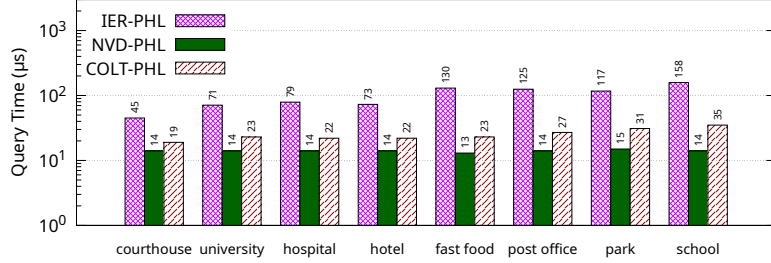

Figure A.18: kNN query performance on different real-world POI sets

Appendix A.1.1. Sensitivity Analysis

Effect of d: With increasing density d, the total number of objects increases, and as a result hierarchical methods like IER-PHL and COLT-PHL scale more poorly than NVD-PHL in Figure A.19a. In this scenario, hierarchical methods are disadvantaged because they must traverse increasingly larger tree paths. Conversely, NVDs need only consider the neighboring Voronoi regions of the NNs found so far, which does not increase significantly (and in fact converges to the average vertex degree with increasing density).

Effect of k: COLT-PHL is only marginally worse than NVD-PHL for varying k in Figure A.19b as it is much less disadvantaged on the default density of 0.001. We note 0.001 corresponds to a more realistic object set size judging by the densities of real-world POI sets in Table 3. The only exception is $k=1$, where NVD-PHL is optimal (the exact 1NN can be retrieved immediately by NVD definition).

Heuristic Efficiency: Given that both methods rely on landmark lower-bounds, COLT-PHL does not need to compute significantly more network distances at lower densities than NVD-PHL as shown in Figure A.19c. While at higher densities lower-bounds become less accurate, NVD-PHL is insulated against this because of the previously mentioned convergence of the number of neighboring Voronoi regions to the average vertex degree with increasing d. The main reason for COLT-PHL's poorer performance is illustrated in Figure A.19d, which shows a significantly higher number of candidate objects are retrieved. This confirms that hierarchical traversals are more susceptible to false candidates in proximity search.
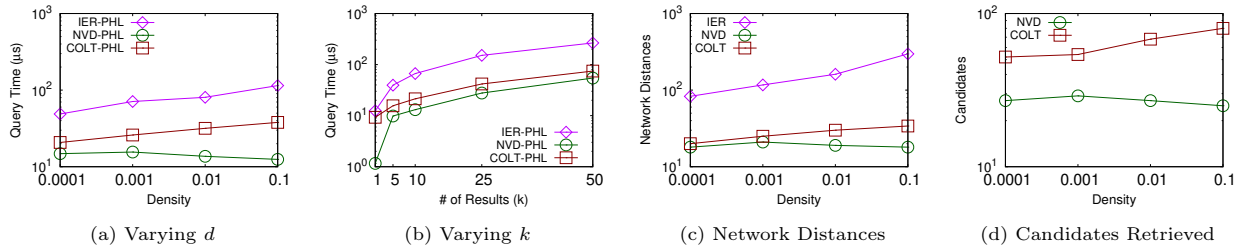

| (a) Varying d | (b) Varying k | (c) Network Distances | (d) Candidates Retrieved |

Figure A.19: Performance for kNN Queries

References

[1] V. Bulitko, Y. Björnsson, R. Lawrence, Case-based subgoaling in real-time heuristic search for video game pathfinding, Journal of Artificial Intelligence Research 39 (2010) 269–300.

[2] D. Papadias, J. Zhang, N. Mamoulis, Y. Tao, Query processing in spatial network databases, in: VLDB, 2003, pp. 802–813.

[3] R. Zhong, G. Li, K.-L. Tan, L. Zhou, Z. Gong, G-tree: An efficient and scalable index for spatial search on road networks, IEEE Trans. Knowl. Data Eng. 27 (2015) 2175–2189.

[4] M. L. Yiu, N. Mamoulis, D. Papadias, Aggregate nearest neighbor queries in road networks, IEEE Trans. Knowl. Data Eng. 17 (2005) 820–833.

[5] S. Wang, M. A. Cheema, X. Lin, Y. Zhang, D. Liu, Efficiently computing reverse k furthest neighbors, in: ICDE, 2016, pp. 1110–1121.

[6] M. Stiglic, N. Agatz, M. Savelsbergh, M. Gradisar, The benefits of meeting points in ride-sharing systems, Transportation Research Part B: Methodological 82 (2015) 36–53.

[7] F. Drews, D. Luxen, Multi-hop ride sharing, in: SoCS, 2013, pp. 71–79.

[8] R. L. Church, Z. Drezner, Review of obnoxious facilities location problems, Computers & Operations Research 138 (2022) 105468.

[9] Y. Gao, L. Shou, K. Chen, G. Chen, Aggregate farthest-neighbor queries over spatial data, in: DASFAA, 2011, p. 149–163.

[10] S. Dasgupta, P. M. Long, Performance guarantees for hierarchical clustering, Journal of Computer and System Sciences 70 (2005) 555–569.

[11] P. Bose, K. Dannies, J.-L. De Carufel, C. Doell, C. Grimm, A. Maheshwari, S. Schirra, M. Smid, Network farthest-point diagrams, Journal of Computational Geometry 4 (2013) 182–211.

[12] D. J. Rosenkrantz, R. E. Stearns, P. M. Lewis, II, An analysis of several heuristics for the traveling salesman problem, SIAM Journal on Computing 6 (1977) 563–581.

[13] T. F. Gonzalez, Clustering to minimize the maximum intercluster distance, Theoretical Computer Science 38 (1985) 293–306.

[14] M. Dyer, A. Frieze, A simple heuristic for the p-centre problem, Operations Research Letters 3 (1985) 285–288.

[15] J. Garcia-Diaz, J. Sanchez-Hernandez, R. Menchaca-Mendez, R. Menchaca-Mendez, When a worse approximation factor gives better performance: a 3-approximation algorithm for the vertex k-center problem, Journal of Heuristics 23 (2017) 349–366.

[16] L. Zhu, Y. Jing, W. Sun, D. Mao, P. Liu, Voronoi-based aggregate nearest neighbor query processing in road networks, in: GIS, 2010, pp. 518–521.

[17] B. Yao, Z. Chen, X. Gao, S. Shang, S. Ma, M. Guo, Flexible aggregate nearest neighbor queries in road networks, in: ICDE, 2018, pp. 761–772.

[18] T. Abeywickrama, M. A. Cheema, Efficient Landmark-Based Candidate Generation for kNN Queries on Road Networks, in: DASFAA, 2017, pp. 425–440.

[19] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: SIGMOD, 1984, pp. 47–57.

[20] A. V. Goldberg, C. Harrelson, Computing the shortest path: A* search meets graph theory, in: SODA, 2005, pp. 156–165.

[21] T. Abeywickrama, M. A. Cheema, S. Storandt, Hierarchical graph traversal for aggregate k nearest neighbors search in road networks, in: ICAPS, 2020, pp. 2–10.

[22] D. Papadias, Q. Shen, Y. Tao, K. Mouratidis, Group nearest neighbor queries, in: ICDE, 2004, pp. 301–312.

[23] M. Goldenberg, N. R. Sturtevant, A. Felner, J. Schaeffer, The compressed differential heuristic, in: AAAI, 2011, pp. 24–29.

[24] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1998) 359–392.

[25] T. Abeywickrama, M. A. Cheema, D. Taniar, k-nearest neighbors on road networks: A journey in experimentation and in-memory implementation, PVLDB 9 (2016) 492–503.

[26] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, S. Zhou, Shortest path and distance queries on road networks: An experimental evaluation, PVLDB 5 (2012) 406–417.

[27] T. Akiba, Y. Iwata, K.-i. Kawarabayashi, Y. Kawata, Fast shortest-path distance queries on road networks by pruned highway labeling, in: ALENEX, 2014, pp. 147–154.

[28] D. Papadias, Y. Tao, K. Mouratidis, C. K. Hui, Aggregate nearest neighbor queries in spatial databases, ACM Trans. Database Syst. 30 (2005) 529–576.

[29] S. Boyd, L. Vandenberghe, Convex Optimization, Cambridge University Press, 2004.

[30] M. Potamias, F. Bonchi, C. Castillo, A. Gionis, Fast shortest path distance estimation in large networks, in: CIKM, 2009, p. 867–876.

[31] D. Šidlauskas, C. S. Jensen, Spatial joins in main memory: Implementation matters!, PVLDB 8 (2014) 97–100.

[32] L. Mérõ, A heuristic search algorithm with modifiable estimate, Artificial Intelligence 23 (1984) 13–27.

[33] T. Abeywickrama, Usa road network graph and points-of-interest dataset, 2025. URL: `https://doi.org/10.5281/zenodo.18026780`.

[34] U. Demiryurek, C. Shahabi, Indexing Network Voronoi Diagrams, in: DASFAA, 2012, pp. 526–543.

[35] T. Abeywickrama, M. A. Cheema, A. Khan, K-spin: Efficiently processing spatial keyword queries on road networks, IEEE Transactions on Knowledge and Data Engineering 32 (2020) 983–997.

[36] T. Abeywickrama, V. Liang, K.-L. Tan, Optimizing bipartite matching in real-world applications by incremental cost computation, PVLDB 14 (2021) 1150–1158.

[37] H. Bast, S. Funke, D. Matijevic, P. Sanders, D. Schultes, In transit to constant time shortest-path queries in road networks, in: WEA, 2007, pp. 46–59.

[38] Z. Chen, B. Yao, Z.-J. Wang, X. Gao, S. Shang, S. Ma, M. Guo, Flexible aggregate nearest neighbor queries and its keyword-aware variant on road networks, IEEE Trans. Knowl. Data Eng. 33 (2021) 3701–3715.

[39] H.-J. Cho, The efficient processing of moving k-farthest neighbor queries in road networks, Algorithms 15 (2022).

[40] M. Chung, S. J. Hyun, W.-K. Loh, Efficient exact k-flexible aggregate nearest neighbor search in road networks using the m-tree, J. Supercomput. 78 (2022) 16286–16302.

[41] P. Ciaccia, M. Patella, P. Zezula, M-tree: An efficient access method for similarity search in metric spaces, in: VLDB, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997, p. 426–435.

[42] H. Wang, K. Zheng, H. Su, J. Wang, S. Sadiq, X. Zhou, Efficient aggregate farthest neighbour query processing on road networks, in: ADC, 2014, pp. 13–25.

[43] Q. T. Tran, D. Taniar, M. Safar, Reverse k Nearest Neighbor and Reverse Farthest Neighbor Search on Spatial Networks, Springer-Verlag, 2009, p. 353–372.

[44] M. Kolahdouzan, C. Shahabi, Voronoi-based k nearest neighbor search for spatial network databases, in: VLDB, 2004, pp. 840–851.

[45] H.-J. Cho, C.-W. Chung, An efficient and scalable approach to cnn queries in a road network, in: VLDB, 2005, pp. 865–876.

[46] Y. Wang, L. Yuan, W. Zhang, Z. Chen, X. Lin, Q. Liu, Simpler is more: Efficient top-k nearest neighbors search on large road networks, PVLDB 17 (2024) 4683–4695.

[47] M. Farhan, H. Koehler, R. Ohms, Q. Wang, Hierarchical cut labelling - scaling up distance queries on road networks, Proc. ACM Manag. Data 1 (2023).

[48] R. Geisberger, P. Sanders, D. Schultes, D. Delling, Contraction hierarchies: Faster and simpler hierarchical routing in road networks, in: WEA, 2008, pp. 319–333.

[49] N. R. Sturtevant, A. Felner, M. Barer, J. Schaeffer, N. Burch, Memory-based heuristics for explicit state spaces, in: IJCAI, 2009, pp. 609–614.

[50] R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, I. Stoica, Beacon vector routing: scalable point-to-point routing in wireless sensornets, in: NSDI, 2005, p. 329–342.

[51] H.-P. Kriegel, P. Kröger, M. Renz, T. Schmidt, Hierarchical graph embedding for efficient query processing in very large traffic networks, in: SSDBM, 2008, pp. 150–167.

[52] A. V. Goldberg, H. Kaplan, R. F. Werneck, Better landmarks within reach, in: WEA, 2007, p. 38–51.

[53] D. C. Rayner, M. H. Bowling, N. R. Sturtevant, Euclidean heuristic optimization, in: AAAI, 2011, pp. 81–86.

[54] M. Bono, A. E. Gerevini, D. D. Harabor, P. J. Stuckey, Path planning with cpd heuristics, in: IJCAI, 2019, pp. 1199–1205.

[55] M. Goldenberg, A. Felner, N. R. Sturtevant, J. Schaeffer, Portal-based true-distance heuristics for path finding, in: SOCS, 2010, pp. 39–45.

[56] L. Cohen, T. Uras, S. Jahangiri, A. Arunasalam, S. Koenig, T. K. S. Kumar, The fastmap algorithm for shortest path computations, in: IJCAI, 2018, pp. 1427–1433.

[57] T. Seidl, H.-P. Kriegel, Optimal multi-step k-nearest neighbor search, in: SIGMOD, 1998, pp. 154–165.

[58] C. Shahabi, M. Kolahdouzan, M. Sharifzadeh, A road network embedding technique for k-nearest neighbor search in moving object databases, GeoInformatica 7 (2003) 255–273.

[59] M. Qiao, L. Qin, H. Cheng, J. X. Yu, W. Tian, Top-k nearest keyword search on large graphs, PVLDB 6 (2013) 901–912.

[60] K. Mouratidis, J. Li, Y. Tang, N. Mamoulis, Joint search by social and spatial proximity, IEEE Trans. Knowl. Data Eng. 27 (2015) 781–793.

[61] S. Yang, M. A. Cheema, X. Lin, W. Wang, Reverse k nearest neighbors query processing: experiments and analysis, PVLDB 8 (2015) 605–616.