Hierarchical Graph Traversal for Aggregate k Nearest Neighbors Search in Road Networks

Tenindra Abeywickrama,[1,2] **Muhammad Aamir Cheema,**[2] **Sabine Storandt**[3]
[1]Grab-NUS AI Lab, National University of Singapore, Singapore
[2]Faculty of Information Technology, Monash University, Melbourne, Australia
[3]Department of Computer and Information Science, University of Konstanz, Konstanz, Germany
tenindra@nus.edu.sg, aamir.cheema@monash.edu, sabine.storandt@uni-konstanz.de

Abstract

Location-based services rely heavily on efficient methods that search for relevant points-of-interest (POIs) close to a given location. A k nearest neighbors (kNN) query is one such example that finds k closest POIs from an agent's location. While most existing techniques focus on finding nearby POIs for a single agent, many applications require POIs that are close to *multiple* agents. In this paper, we study a natural extension of the kNN query for multiple agents, namely, the Aggregate k Nearest Neighbors (AkNN) query. An AkNN query retrieves k POIs with the smallest aggregate distances where the aggregate distance of a POI is obtained by aggregating its distances from the multiple agents (e.g., sum of its distances from each agent). Existing search heuristics are designed for a single agent and do not work well for multiple agents. We propose a novel data structure COLT (Compacted Object-Landmark Tree) to address this gap by enabling efficient hierarchical graph traversal. We then utilize COLT for a wide range of aggregate functions to efficiently answer AkNN queries. In our experiments on real-world and synthetic data sets, our techniques significantly improve query performance, typically outperforming existing approaches by more than an order of magnitude in almost all settings.

1 Introduction

Finding nearby relevant objects, and in particular points-of-interest (POIs), efficiently is a critical task in a variety of planning and scheduling applications including, e.g., real-time path planning in video maps (Bulitko, Björnsson, and Lawrence 2010) or location-based services. Nearby relevant POIs are typically obtained by using a range query or a k Nearest Neighbors (kNN) query. A range query (Papadias et al. 2003) returns all POIs within a given distance from an agent's location, e.g., find all restaurants within 1 km from a user. A kNN query (Zhong et al. 2015) returns the k closest POIs from an agent's location, e.g., find three fuel stations closest 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 measure of proximity in many real-world applications such as map-based services and is capable of representing a variety of metrics such as physical distance, travel time, toll cost, etc.

While the aforementioned queries retrieve POIs that are close to a single agent, in many applications (e.g., ride-sharing (Stiglic et al. 2015; Drews and Luxen 2013)), it is important to obtain nearby POIs considering the locations of multiple agents. An aggregate k nearest neighbors (AkNN) query (Yiu, Mamoulis, and Papadias 2005) is a natural extension of a kNN query for multiple agents that retrieves POIs considering their aggregate distance from the 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 k POIs with the smallest aggregate distances. 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 the ease of presentation, 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).

AkNN queries have many real-world applications. Consider a group of friends planning to meet at a restaurant. They might 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 distance of the station from the furthest residential area 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. While we focus on AkNN queries in road networks, it is worth noting that AkNN 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.

The AkNN queries can also be used to solve another important query named a *shortest detour query*. 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 the example of a user who wants to stop at a gas station on the way from her home to work. In this case, a shortest detour query helps find the gas station that minimizes the total distance she needs to travel. Note that a shortest detour query is a special case of an AkNN query where s and t are considered to be the query locations and the aggregate function is sum.

Limitations of Existing Techniques

The key to efficiently retrieving nearby POIs in road networks is in developing heuristics to find the most promising POI candidates. Heuristics to answer AkNN queries have largely been borrowed from kNN techniques (Yiu, Mamoulis, and Papadias 2005; Zhu et al. 2010; Yao et al. 2018), which is problematic in several ways. First, intuitions to find kNNs do not necessarily translate to AkNNs. For example, the most efficient kNN heuristic (Abeywickrama and Cheema 2017) 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 1st nearest POI of every query location and the adjacency relationships between 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.

Given this, a more appropriate heuristic to find AkNNs is to conduct a hierarchical search on the road network. (Yiu, Mamoulis, and Papadias 2005) search an R-tree (Guttman 1984) containing the POIs in a top-down manner according to a Euclidean distance heuristic, similar to another kNN technique (Papadias et al. 2003). When constructed, the R-tree 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.

Landmark Lower-Bounds (LLBs) are a more accurate alternative to Euclidean distance (Goldberg and Harrelson 2005). 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. However, there is no hierarchical data structure to compute minimum LLBs to groups of POIs in the same way as R-trees using Euclidean distance.

Contributions

We propose techniques to overcome the diverse challenges mentioned above. Our contributions are summarized below.

- We present two hierarchical data structures, SL-Tree and COLT. These add significantly more landmarks than previous methods, while still keeping the data structures reasonably small in theory and practice. COLT, the main index used for querying, is particularly light-weight. It is also the first index to support hierarchical traversal of the road network to locate POIs by landmark lower-bounds.
- Utilizing COLT and part of the SL-Tree, we propose a heuristic search algorithm to efficiently answer AkNN queries. Our algorithm is particularly adept at AkNN queries due to a unique property of COLT for convexity-preserving aggregate functions such as sum and max.
- We demonstrate the significant improvement in query time and heuristic efficiency of our techniques on real-world datasets through extensive experiments, achieving up to two to three orders of magnitude improvement.

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., 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 on graph vertices for simpler exposition.

Aggregate k Nearest Neighbor (AkNN) Queries: Given a set of query vertices $Q \in V$, a set of object vertices $P \in V$, 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 agg function is either sum or max. We remark that AkNN queries are also known as Group kNN queries (Papadias et al. 2004).

Landmark Lower-Bounds (LLBs): Lower-bounds based on landmarks, also called differential heuristics (Goldenberg et al. 2011), 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 (1) using the distances to landmark l_i and the triangle inequality. The maximum lower-bound over all m landmarks given by (2) gives a tighter lower-bound and is typically accurate even for small m (Goldberg and Harrelson 2005).

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

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

Lower-Bound Aggregate Distance: (Yiu, Mamoulis, and Papadias 2005) showed, 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 $d_{agg}(Q, p)$, as in Lemma 1.

Lemma 1. *Given a monotonic aggregate function agg and lower-bounds distances $LB(q_i, p)$ 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., $LB_{agg}(Q, p) = agg(LB(q_0, p), \dots, LB(q_{|Q|}, p)) \leq d_{agg}(Q, p)$.*

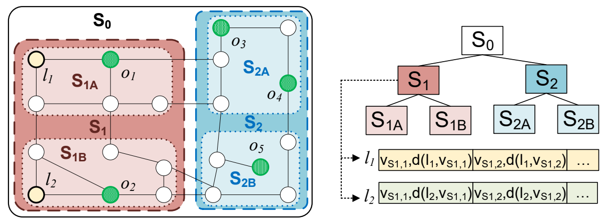
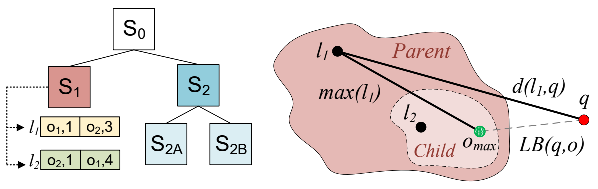

S0
S1A
S1
S1B
S2A
S2
S2B
l1
l2
o1
o2
o3
o4
o5
l1 v_{S1,1},d(l1,v_{S1,1}) v_{S1,2},d(l1,v_{S1,2}) ...
l2 v_{S1,1},d(l2,v_{S1,1}) v_{S1,2},d(l2,v_{S1,2}) ...
(a) SL-Tree Subgraph Partitions (b) SL-Tree Node Hierarchy

Figure 1: Subgraph Landmark Tree (SL-Tree)



(a) COLT Hierarchy (b) COLT Lower-Bounds

Figure 2: Compacted Object-Landmark Tree (COLT)

3 Data Structures

We now describe our data structures to index the road network and object set for efficient hierarchical graph traversal.

Road Network Index: We first introduce the Subgraph-Landmark Tree (SL-Tree) to index the road network. The SL-Tree is a supporting index that we use to construct our object index efficiently. Each node in the SL-Tree represents a subgraph of the road network with the root being G. 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 SL-Tree shown in Figure 1b. Note that we refer to tree *nodes* and road network *vertices*.

For each node n_T, we select m of its vertices as *local landmarks*, e.g., l_1 and l_2 for $m = 2$ for S_1 in Figure 1a (landmarks for other nodes are omitted for clarity). We then compute distances from each landmark l_i to every vertex in n_T's subgraph using Dijkstra's search, which are then stored in *distance list* DL_i. Figure 1b shows the distance lists for the landmarks of S_1 (those for other nodes are again omitted). Note that subgraphs are stored implicitly by mapping road network vertices to the SL-Tree leaf nodes that contains them. Any technique capable of partitioning graphs into equally sized subgraphs can be used (the only partitioning constraint), e.g., METIS (Karypis and Kumar 1998).

Object Index: The SL-Tree can then be used to efficiently construct our object index, the Compacted Object-Landmark Tree (COLT). COLT is a carefully compacted version of the SL-Tree for object set P. Compaction ensures that there are m local landmarks for at most λ objects, to increase the likelihood of finding a tighter lower-bound for more objects. Note that the SL-Tree is shared between construction of all COLT indexes, i.e., for many different object sets.

Given SL-Tree T, COLT index 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 in a leaf node for COLT. If n_T contains more than λ objects, the search expands to its children. Otherwise, the search is pruned at n_C, which becomes a leaf-node of COLT. For the new leaf n_C, an *Object Distance List* ODL_i is created in n_C for each landmark l_i in n_T. These are simply the distance lists of n_T, except with only the distances for object vertices from P. Any interior nodes with only one child are merged with the child (keeping the child's more localized landmarks). Figure 2a shows a COLT index for $\lambda = 2$ constructed from the SL-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 not shown 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 object distance list ODL_i of leaf node n_{leaf} in C is sorted on 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 of its landmarks l_i. These are computed using distance lists in corresponding SL-Tree nodes. Next, we use this information to compute lower-bounds to nodes and traverse the hierarchy.

Lower-Bound Heuristic for Graph Traversal

Similar to the lower bound in (1), we can compute a lower bound for all objects contained within a node n_C in COLT index C by (3) for one landmark $l_i \in n_C$. (4) gives the best lower-bound over all m landmarks of n_C:

$$LB_{l_i}(n_C, q) = \begin{cases} d(l_i, q) - M^+ & \text{if } d(l_i, q) \geq M^+ \\ M^- - d(l_i, q) & \text{if } d(l_i, q) \leq M^- \\ 0 & \text{else} \end{cases} \quad (3)$$

$$LB_{max}(n_C, q) = \max_{l_i \in n_C} (LB_{l_i}(n_C, q)) \quad (4)$$

Thereby, $M^- := min_{n_C,l_i}$ and $M^+ := max_{n_C,l_i}$ for n_C are already available in COLT. However, for non-root nodes, $d(l_i, q)$ in (3) 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. Alternatively, computing $d(l_i, q)$ on the fly using another technique is expensive and may be wasteful if the node does not contain results. Interestingly, (3) still holds if we replace the distances with lower-bound $LB(l_i, q)$ and upper bound $UB(l_i, q)$, as in (5). The distance lists of the root or lowest common ancestor SL-Tree node can be conveniently used to compute the best $LB(l_i, q)$ and $UB(l_i, q)$ by (2) and its upper-bound equivalent (by adding rather than subtracting distances), respectively. Choosing the tightest over all landmarks of n_C gives an inexpensive and accurate bound even for a small number of landmarks:

$$LB_{l_i}(n_C, q) = \begin{cases} LB(l_i, q) - M^+ & \text{if } LB(l_i, q) \geq M^+ \\ M^- - UB(l_i, q) & \text{if } UB(l_i, q) \leq M^- \\ 0 & \text{else} \end{cases} \tag{5}$$

Landmark Choices: Additional landmarks closer to the objects naturally lead to more accurate lower-bounds. Past work has shown that landmarks on the fringe of the graph also give better lower-bounds (Goldberg and Werneck 2005). Inspired by this, we choose landmarks from *border* vertices of the subgraph, i.e., vertices with an edge leading to a vertex in a different subgraph. First, we partition the plane into m equally sized slices around the Euclidean center of the subgraph, as if we were cutting a cake. We choose 1 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.

Effective Lower-Bounds: The accuracy of COLT's inexpensive lower-bounds increases as we delve deeper into the hierarchy. In Figure 2b, let us say we use l_1 and its maximum object distance to compute a lower-bound for the child 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. Next, we utilize this as a decoupled heuristic for A*k*NN querying.

4 Query Algorithm for A*k*NN Search

Recent studies (Yao et al. 2018; Abeywickrama, Cheema, and Taniar 2016) have shown that *decoupled heuristics* are highly effective for POI search in road networks. Accordingly, we utilize COLT to retrieve candidate objects (POIs) likely to be A*k*NN results, by their lower-bounds and then use another technique to compute network distances. While iteratively retrieving further candidates, the set of the best k candidates are updated, terminating when it can no longer be improved. Interestingly, COLT considers fewer A*k*NN candidates to find the object with the minimum lower-bound using a novel property of Object Distance Lists (ODLs).

Object Distance Lists and Convexity

Note that (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 Object Distance List ODL for the landmark l for a single query vertex q. Since ODL essentially stores the domain of x for all objects in the node, the minimum lower-bound for the landmark l can be found by searching ODL for $d(l, c)$ closest to $d(l, q)$ for some object $c \in ODL$. Since ODL is sorted, this is possible using a modified binary search, as observed by (Abeywickrama and Cheema 2017), in only $O(\log \lambda)$ time.

Finding the minimum lower-bound *aggregate* distance 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 (Yiu, Mamoulis, and Papadias 2005). 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, this might suggest we need to search ODL for multiple values (i.e., once 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, we find that the most widely used functions (Papadias et al. 2005), max and sum, do preserve convexity. (Boyd and Vandenberghe 2004) prove convexity preservation for a range of functions.

Specifically, once the minimum x^* of 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 ODL, due to the convexity of the function. 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 two 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 $\frac{C_{min} + C_{max}}{2}$. □

Query Processing

Algorithm 1 uses hierarchical graph traversal on the COLT index to guide us towards ODLs most likely to contain A*k*NN results. The algorithm maintains a priority queue $\mathcal{PQ}$ containing objects and COLT 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 is updated (lines 7-10). The result set R

Algorithm 1 Get AkNNs by COLT for query vertex set Q

```
1: function GETAKNNS($k, Q, agg, COLT, SLTree$)
2:     $\mathcal{PQ} \leftarrow \phi$  ▷ Priority queue keyed by lower-bound distance
3:     $R \leftarrow \phi$  ▷ Max priority queue containing $k$ best candidates
4:     INSERT($\mathcal{PQ}, COLT.root, 0$) , $D_k \leftarrow \infty$
5:     while MINKEY($\mathcal{PQ}$) $< D_k$ and $\mathcal{PQ}$ not empty do
6:         $c \leftarrow$ EXTRACT-MIN($\mathcal{PQ}$)
7:         if $c$ is an object then
8:             Compute agg. dist $d_{agg}(Q, c)$ using $d(q_i, c) \forall q_i \in Q$
9:             if $d_{agg}(Q, c) < D_k$ then
10:                INSERT($R, c, d_{agg}(Q, c)$)  ▷ Check/update $R$ & $D_k$
11:        else if $c$ is a non-leaf node, i.e., with no $ODL$ then
12:            for each child node $e$ of $c$ do
13:                Compute $LB_{agg}(Q, e)$ for $e$ by (4) and Lemma 1
14:                INSERT($\mathcal{PQ}, e, LB_{agg}(Q, e)$)
15:        else if $c$ is a leaf node then
16:            if $c$ not seen before then
17:                Set $c.l$ to furthest landmark by LLB from $Q$
18:                Compute network distances $d(q, c.l) \forall q \in Q$
19:                Initialize array $c.d[]$ using query distances to $c.l$
20:                Binary search $c.l$'s $ODL$ for minimum of $f(x)$
21:                Initialize $c.RP$/$c.LP$ to index of minimum
22:            RETRIEVEOBJECTSOL($\mathcal{PQ}, Q, c$)
23:            Set best lower-bound $LB_{agg}(Q, c)$ using $c.RP$/$c.LP$
24:            INSERT($\mathcal{PQ}, c, LB_{agg}(Q, c)$)
25:    return $R$
26: function RETRIEVEOBJECTSOL($\mathcal{PQ}, Q, c$)
27:    while $LB_{agg,c.l}(Q, p) < \mathcal{PQ}.Top()$ do
28:        $p \leftarrow$ object at $c.RP$ or $c.LP$ with smaller $LB_{agg,c.l}$
29:        INSERT($\mathcal{PQ}, p, LB_{agg,max}(Q, p)$)
30:        Increment $c.RP$ or decrement $c.LP$ used at Line 28
```

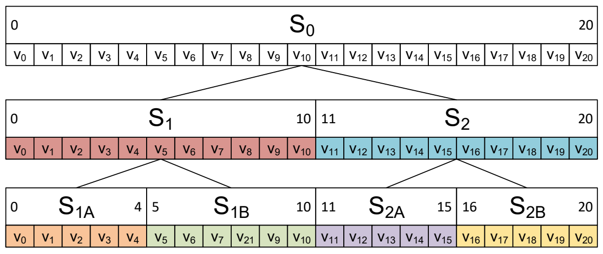
maintains up to k objects with the smallest aggregate distances found so far by the algorithm and D_k corresponds to the k-th largest aggregate distance among these objects. If a non-leaf node is extracted then an aggregate lower-bound score is computed according to (4) and (5) for each of its child nodes (lines 12-14). If it is a leaf-node, it is initialized if encountered for the first time (lines 16-21). A landmark is chosen to determine the object list to process, 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 RETRIEVEOBJECTSOL 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. We set $c.l$ (line 17) to the landmark furthest from the query vertices (on average) by a lower-bound computed using the SL-Tree and (2), to obtain tighter lower-bounds similar to fringe landmarks in Section 3. Line 29 computes the maximum (best) lower-bound aggregate distance for object p using available network distances for any landmark (e.g., landmarks in the root SL-Tree node or $c.l$ in the leaf).

Proof Sketch for Correctness: The intuition behind the algorithm's correctness is that there is a lower-bound aggregate distance for every object in the priority queue. That is, a queue element for the object, or a queue element for a subgraph containing the object(s). Thus, when the algorithm terminates, lower-bounds for all remaining objects are greater than 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.

5 Complexity Analysis

In this section, we analyze the space and time complexity of the proposed data structures COLT and SL-Tree. COLT is converted from the SL-Tree, which is a complete b-ary tree. There are at most $O(|P|)$ leaf nodes in COLT. COLT may be unbalanced as not all nodes in the SL-Tree may be included in COLT. However, since we merge child nodes into their parent nodes if the child does not have other siblings (compressing the tree height), the total number of nodes will be the same as a complete b-ary tree, i.e., $O(|P|)$. However, COLT's space complexity is dominated by the m object distance lists (ODLs) stored in each leaf node. This results in $O(m|P|)$ total space as each object is only stored in one ODL and we remark that m is a small constant. Similarly, compression of COLT's height via merging of single child nodes into parent nodes results in an average depth in COLT of $O(\log |P|)$. Given the top-down conversion and $O(1)$ look-ups of SL-Tree vertex distances lists as hash-tables, propagating $|P|$ objects to build ODLs and computing required values takes $O(|P| \log |P|)$ time. Sorting all ODLs takes $O(|P| \lambda \log \lambda)$ time where λ is a small constant. Thus, the total time complexity is $O(|P| \log |P|)$.

The height of the balanced SL-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 vertex distance lists associated with the SL-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 and vertex-leaf node mapping. 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 SL-Tree node's subgraph, e.g., if a shortest path from a landmark to one vertex passes outside the subgraph. We multiply b^i by a factor γ to indicate how many additional vertices are visited by the search. In practice, γ can be reduced by using techniques to limit the search. A simple method is to use a bounding box around the subgraph and use an A*-like lower-bounding heuristic to prune the Dijkstra's search. More complex techniques like border-to-border shortcuts (Lee, Lee, and Zheng 2009) or distance matrices (Zhong et al. 2015) can reuse distance computations from lower-levels to reduce the graph at higher levels and accelerate computation. We find that γ is a small constant on average in practice, and the overall time complexity for level i simplifies to $O(m|V| \log |V|)$. Multiplying by the height, the time complexity for the whole tree is $O(m|V|(\log |V|)^2)$. Thus, space and time cost only increase by a factor of $O(\log |V|)$ over ALT.



S0
0
20
v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18 v19 v20
S1
S2
0 10 11 20
S1A S1B S2A S2B
0 4 5 10 11 15 16 20
v0 v1 v2 v3 v4 v5 v6 v7 v21 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18 v19 v20

Figure 3: Hierarchical Subgraph Vertex Ordering

Parameter	Values
k	1, 5, **10**, 25, 50
d	1, 0.1, 0.01, **0.001**, 0.0001
A (%)	1, 5, **15**, 50, 100
$\vert Q\vert$	2, 4, **8**, 16, 32
Agg. Func.	*sum*, ***max***
Real-World POI Set ($\vert P\vert$)	School (160,525), Park (69,338), Fast Food (25,069), Post Office (21,319), Hospitas (11,417), Hotels (8,742), University (3,954), Courthouse (2,161)

Table 1: Parameters (Defaults in bold if applicable)

Subgraph Vertex Ordering: It is not practicable in terms of space cost to store the road network vertices associated with each SL-Tree node (i.e., belonging to the subgraph associated with the SL-Tree node). This would involve storing the same road network vertex at each level of the tree. Consequently, vertices are only stored in the leaf node of the SL-Tree. However, the construction time complexity analysis given earlier depends on a top-down conversion of the SL-Tree structure into a COLT tree. This involves determining which child SL-Tree node a road network vertex belongs to in $O(1)$ time. With implicit storage of vertices in non-leaf nodes, this look-up would take $O(log|V|)$ time, by first finding the leaf node containing the vertex and following the hierarchy upwards. This can be avoided by an ordering which groups vertices of child subgraphs recursively as shown in Figure 3 and renumbering vertex IDs based on this order. By simply storing the first and last vertex ID of vertices in a child subgraph, point containment can be performed in constant time $O(1)$. As an added benefit of subgraph vertex ordering, we reduce the space of the SL-Tree distances lists by half as they need only store distances with the list in subgraph-vertex order. For very large road networks this can entail gigabytes of storage saved. Moreover, distance lists can be stored as arrays instead of hash-tables, which consume a great deal more memory than just its elements and are known to incur performance penalties due to poor data locality (Šidlauskas and Jensen 2014; Abeywickrama, Cheema, and Taniar 2016).

6 Experiments

Experimental Settings

Environment: We conduct experiments on a Linux (64-bit) Amazon Web Services r5a.2xlarge instance with eight AMD EPYC 2.5GHz CPU cores and 64GB of memory. Code was written in single-threaded C++ and compiled by g++ v5.4 with O3 flag. All experiments were conducted using memory-resident indexes for fast query processing.

Datasets: We use a real road network graph for the continental US with $23,947,347$ vertices and $57,708,624$ travel time edges obtained from the 9th DIMACS Challenge[1] combined with 8 real POI sets for the US from OSM[2], as listed in Table 1 provided by (Abeywickrama, Cheema, and Taniar 2016). For sensitivity analysis, we generate synthetic POI sets chosen uniformly at random for density d where $d{=}|P|/|V|$.

Parameters: We use parameter A to define a connected subgraph of G with $A\%$ of the total vertices $|V|$. Query vertices are then chosen uniformly at random from the $A\%$ subgraph. Similar to past studies (Yiu, Mamoulis, and Papadias 2005), this represents how "local" a group of query locations are. We test the sensitivity of techniques to varying numbers of results k, density d, query vertices $|Q|$, and subgraph percentages A. We also test two popular aggregate functions max and sum. Parameter values are listed in Table 1 with defaults in bold if applicable. We report query time over 500 queries, e.g., with 10 sets of objects and 50 sets of randomly chosen query vertices for each object set.

Techniques: We compare our algorithm against the Incremental Euclidean Restriction (IER) AkNN algorithm proposed by (Yiu, Mamoulis, and Papadias 2005). We also apply their concurrent expansion approach to adapt the state-of-the-art kNN heuristic based on Network Voronoi Diagrams (NVDs) by (Abeywickrama and Cheema 2017) for AkNN queries. Each technique uses Pruned Highway Labeling (PHL) (Akiba et al. 2014) implemented by its authors to compute network distances, hence techniques are named IER-PHL, NVD-PHL, and COLT-PHL. Using the same network distance computation technique ensures a level playing field, and as PHL is one of the fastest techniques, it will better show the differences in overheads. We implemented all other techniques, sharing subroutines and basic data structures to ensure fairness. Nonetheless, we also compare algorithms on heuristic performance in terms of "false positives", which is independent of the network distance technique. SL-Trees and COLT use branching factor $b = 4$, maximum object distance list size $\lambda = 1024$ (which is also α)

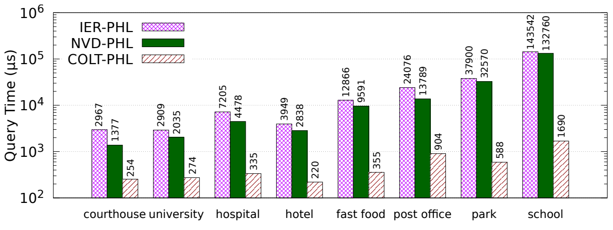
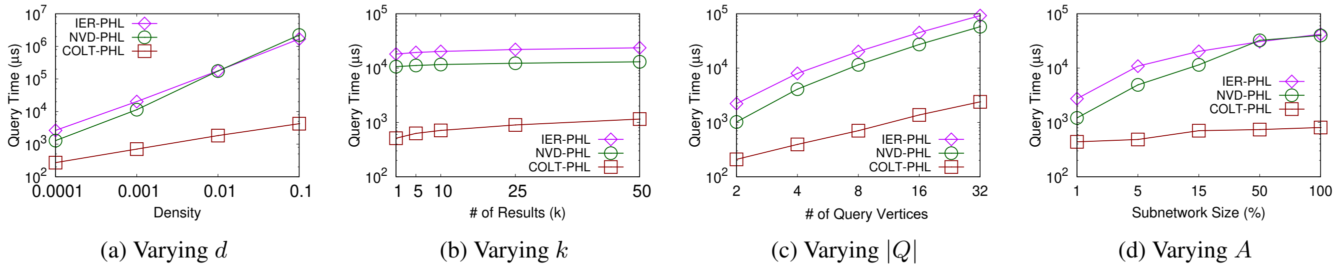


Figure 4: Performance on different real-world POI sets

[1]http://www.dis.uniroma1.it/%7Echallenge9/
[2]http://www.openstreetmap.org



(a) Varying d (b) Varying k (c) Varying $|Q|$ (d) Varying A

Figure 5: Performance for max function

and $m = 4$ landmarks per node for ideal performance vs. index size. NVD uses the ALT index (Goldberg and Harrelson 2005) with $m = 16$ random landmarks to compute LLBs, which we also use as it is essentially the root of an SL-Tree.

Real-World Query Performance

Figure 4 depicts query time on real-world POI datasets, with the number of objects increasing from left to right. COLT significantly outperforms the other methods across the board, with up to two orders of magnitude improvement. COLT tends to improve more on larger POI sets, where it is more difficult to distinguish between objects. Next, our sensitivity analysis delves deeper into this and other nuances of query performance for varying parameters.

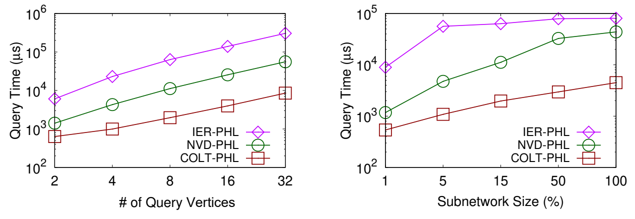
Sensitivity Analysis

Effect of d: The trend seen for real-world POIs is confirmed by increasing object density d in Figure 5a. Both IER and NVD scales 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's tighter lower bounds and hierarchical traversal is more effective in pruning subgraphs and pinpointing likely candidates.

Effect of k: COLT significantly outperforms the other methods for varying k in Figure 5b. NVD-PHL and IER-PHL query times do not vary significantly compared to COLT (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, NVD-PHL expands so many candidates to find the first AkNN, that subsequent candidates have already been encountered.

Effect of $|Q|$: We investigate query time as the number of query vertices increases in Figure 5c. Increasing query vertices involves computing additional lower-bounds and network distances to candidates, thus query time increases for all methods. However, COLT scales better because it conducts a single binary search on its object distance lists 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.

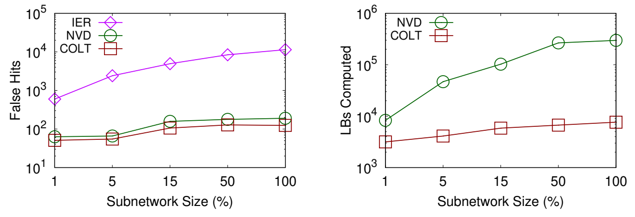
Effect of A: Recall that A is the percentage of graph vertices in a subgraph from which we choose query vertices. With increasing A query vertices become further apart, e.g., to represent a query by a logistics company placing depots nation-wide. For $A = 1\%$, NVD-PHL performance is closer to COLT in Figure 5d. For NVDs, this scenario is similar to kNN queries where it excels, as more query vertices share the same 1NN and concurrent expansions overlap. IER does not benefit from this as its lower-bounds are still inaccurate. In a sense, queries become "harder" with increasing A and COLT scales extremely well in that case.



(a) Varying $|Q|$ (b) Varying A

Figure 6: Performance for sum function

Effect of Aggregate Function: We evaluate another popular aggregate function, sum, in Figure 6. Both IER and COLT have higher query times for sum than max because sum also sums the error of the lower-bound. This effect is amplified by the hierarchical nature of IER and COLT, explaining the relative improvement of NVD-PHL. However, COLT's tighter lower-bounds make it more robust to this than IER and still significantly outperforms both methods.



(a) False Hits (b) Landmark Lower-Bounds

Figure 7: Heuristic performance

Heuristic Efficiency: We measure the efficiency of the heuristics in Figure 7 using machine-independent metrics. Figure 7a shows the number of network distances computed

	ALT (m=16)	SL-Tree (m=4)	PHL
Time	71s	25m	25m
Space	1.43GB	4.6GB	15.8GB

Table 2: Road Network Index Statistics (US)

	COLT (m=4)	NVD	R-tree
Time	63ms	11s	6ms
Space	0.9MB	28MB	0.9MB

Table 3: Object Index Statistics (US,uniform objects,d=0.001)

to non-result POIs. The poor query time performance of IER is explained by the significantly higher network distances computed. However, NVDs do not compute much more network distances than COLT. As both methods use landmark lower-bounds (LLBs), that are more accurate than IER's Euclidean distance, they both avoid computing network distances. The poor query time of NVDs can be explained by Figure 7b, which shows NVDs computing a significantly higher number of LLBs than COLT. This confirms expansion in NVDs encounters a significantly higher number of objects than COLT's hierarchical search.

Pre-Processing Costs

Table 2 details the road network pre-processing costs in terms of time and space. SL-Tree consumes greater space than ALT, but not significantly so. Moreover, only the root node of the SL-Tree is required for query processing, which has the same index size as ALT. The indexing time is comparable to PHL, which possesses one of the fastest pre-processing times for high-performance indexes (Akiba et al. 2014). Table 3 lists the pre-processing costs for object indexes used by each technique for default density $0.001 \times |V|$. Note that object indexes are constructed for each object set (e.g., the set of restaurants). COLT is significantly smaller and faster to construct than NVDs and is comparable to R-trees as both have space complexity linear to the input.

7 Related Work

A recent experimental study (Abeywickrama, Cheema, and Taniar 2016) on the kNN problem provided an in-depth review of the state-of-the-art. The key findings of this study were the surprising performance of IER and the implications this had on heuristics used in kNN. We refer the reader to this paper for a detailed review of kNN techniques.

One of the most popular heuristics to estimate network distances is landmark-based lower-bounds (LLBs) (Goldberg and Harrelson 2005) also often referred to as differential heuristics (Sturtevant et al. 2009; Goldenberg et al. 2011). (Kriegel et al. 2008) proposes a hierarchical landmark scheme to reduce index space cost. Other work (Goldberg and Werneck 2005) has focused on improving lower bounds, e.g., through better landmark selection. These compliment our work, e.g., better lower bounds reduce the number of false hits. Other notable heuristics include Euclidean distance-based heuristics (Rayner, Bowling, and Sturtevant 2011), compressed path databases (Bono et al. 2019), portal-based true-distance (Goldenberg et al. 2010), and FastMap (Cohen et al. 2018) heuristics.

Landmarks have been used to answer kNN queries by (Kriegel et al. 2007; 2008) based on the multi-step kNN paradigm (Seidl and Kriegel 1998). While these studies propose interesting improvements to using landmarks, the kNN algorithms require creating a ranking by computing lower-bounds to all objects. As discussed, this approach is not scalable with object set size and not competitive with existing approaches in practice. VN^3 (Kolahdouzan and Shahabi 2004) uses Network Voronoi Diagrams to answer kNN queries. NVD-based techniques were also used to answer continuous kNN queries (Zheng et al. 2016). (Mouratidis et al. 2015) proposed computing landmark lower-bounds to groups of users in a social network similar to (3). However, their technique is not designed for hierarchical graph traversals and, unlike COLT, does not address the problem of computing more accurate lower-bounds (e.g., with more landmarks) necessary for an effective hierarchical search.

Road Network Embedding (Shahabi, Kolahdouzan, and Sharifzadeh 2003) transforms the road network into higher dimensional space and uses Minkowski metrics to estimate network distance. However, the proposed kNN method is approximate. Qiao et al. also propose an approximate technique (Qiao et al. 2013) using shortest path trees to compute distance estimates based on tree distance, but their solution applies to the *keyword* search problem in road networks.

The AkNN problem on road networks was addressed by applying IER (Yiu, Mamoulis, and Papadias 2005). However, this experiences the same inaccuracy problems as IER in kNN search, especially when edge-weights are not physical distance. In fact, AkNN search is more adversely affected as it also aggregates the error. VN^3 has also been applied to the AkNN problem (Zhu et al. 2010), but like the kNN algorithm, it suffers from high pre-processing costs.

8 Conclusion

COLT elegantly combines several properties that benefit AkNN search. First, AkNN queries involve multiple query locations. Consequently, result objects are unlikely to be near any query location and are more easily located using the hierarchical subgraph traversal in COLT. Second, COLT can compute better and inexpensive lower-bounds using localized landmarks at each level of the hierarchy. Combined with its novel property for convexity preserving aggregate functions, we can retrieve more promising candidates and terminate search sooner. This is demonstrated in our experiments with COLT significantly outperforming competing techniques on AkNN queries. Moreover, the data structures used for querying are light-weight in both theory and practice. COLT is also a versatile data structure with potential application to other search and planning problems, e.g., utilizing the hierarchical search to find optimal meeting points for ride-sharing. Further improvement of search heuristics may also be possible, e.g., in the direction of Compressed Differential Heuristics (Goldenberg et al. 2011).

Acknowledgments

Muhammad Aamir Cheema is supported by Australian Research Council FT180100140 and DP180103411. This work was supported by the German Research Foundation (SPP 1894, contract no. STO 1114/4-2) and an Australian Government RTP Scholarship. The work was mainly completed while Tenindra Abeywickrama was with Monash University.

References

Abeywickrama, T., and Cheema, M. A. 2017. Efficient Landmark-Based Candidate Generation for kNN Queries on Road Networks. In *DASFAA*, 425–440.

Abeywickrama, T.; Cheema, M. A.; and Taniar, D. 2016. k-nearest neighbors on road networks: A journey in experimentation and in-memory implementation. *PVLDB* 9(6):492–503.

Akiba, T.; Iwata, Y.; Kawarabayashi, K.-i.; and Kawata, Y. 2014. Fast shortest-path distance queries on road networks by pruned highway labeling. In *ALENEX*, 147–154.

Bono, M.; Gerevini, A. E.; Harabor, D. D.; and Stuckey, P. J. 2019. Path planning with CPD heuristics. In *IJCAI*.

Boyd, S., and Vandenberghe, L. 2004. *Convex Optimization*. Cambridge University Press.

Bulitko, V.; Björnsson, Y.; and Lawrence, R. 2010. Case-based subgoaling in real-time heuristic search for video game pathfinding. *Journal of Artificial Intelligence Research* 39:269–300.

Cohen, L.; Uras, T.; Jahangiri, S.; Arunasalam, A.; Koenig, S.; and Kumar, T. K. S. 2018. The fastmap algorithm for shortest path computations. In *IJCAI*.

Drews, F., and Luxen, D. 2013. Multi-hop ride sharing. In *Sixth annual symposium on combinatorial search*.

Goldberg, A. V., and Harrelson, C. 2005. Computing the shortest path: A* search meets graph theory. In *SODA*, 156–165.

Goldberg, A. V., and Werneck, R. F. F. 2005. Computing point-to-point shortest paths from external memory. In *ALENEX*, 26–40.

Goldenberg, M.; Felner, A.; Sturtevant, N. R.; and Schaeffer, J. 2010. Portal-based true-distance heuristics for path finding. In *SOCS*.

Goldenberg, M.; Sturtevant, N. R.; Felner, A.; and Schaeffer, J. 2011. The compressed differential heuristic. In *AAAI*.

Guttman, A. 1984. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 47–57.

Karypis, G., and Kumar, V. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 20(1):359–392.

Kolahdouzan, M., and Shahabi, C. 2004. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB*, 840–851.

Kriegel, H.-P.; Kröger, P.; Kunath, P.; and Renz, M. 2007. Generalizing the optimality of multi-step k-nearest neighbor query processing. In *SSTD*, 75–92.

Kriegel, H.-P.; Kröger, P.; Renz, M.; and Schmidt, T. 2008. Hierarchical graph embedding for efficient query processing in very large traffic networks. In *SSDBM*, 150–167.

Lee, K. C. K.; Lee, W.-C.; and Zheng, B. 2009. Fast object search on road networks. In *EDBT*, 1018–1029.

Mouratidis, K.; Li, J.; Tang, Y.; and Mamoulis, N. 2015. Joint search by social and spatial proximity. *IEEE Trans. Knowl. Data Eng.* 27(3):781–793.

Papadias, D.; Zhang, J.; Mamoulis, N.; and Tao, Y. 2003. Query processing in spatial network databases. In *VLDB*, 802–813.

Papadias, D.; Shen, Q.; Tao, Y.; and Mouratidis, K. 2004. Group nearest neighbor queries. In *ICDE*, 301–312.

Papadias, D.; Tao, Y.; Mouratidis, K.; and Hui, C. K. 2005. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.* 30(2):529–576.

Qiao, M.; Qin, L.; Cheng, H.; Yu, J. X.; and Tian, W. 2013. Top-k nearest keyword search on large graphs. *PVLDB* 6(10):901–912.

Rayner, D. C.; Bowling, M. H.; and Sturtevant, N. R. 2011. Euclidean heuristic optimization. In *AAAI*.

Seidl, T., and Kriegel, H.-P. 1998. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, 154–165.

Shahabi, C.; Kolahdouzan, M.; and Sharifzadeh, M. 2003. A road network embedding technique for k-nearest neighbor search in moving object databases. *GeoInformatica* 7(3):255–273.

Stiglic, M.; Agatz, N.; Savelsbergh, M.; and Gradisar, M. 2015. The benefits of meeting points in ride-sharing systems. *Transportation Research Part B: Methodological* 82:36–53.

Sturtevant, N. R.; Felner, A.; Barer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *IJCAI*, 609–614.

Šidlauskas, D., and Jensen, C. S. 2014. Spatial joins in main memory: Implementation matters! *PVLDB* 8(1):97–100.

Yao, B.; Chen, Z.; Gao, X.; Shang, S.; Ma, S.; and Guo, M. 2018. Flexible aggregate nearest neighbor queries in road networks. In *ICDE*, 761–772.

Yiu, M. L.; Mamoulis, N.; and Papadias, D. 2005. Aggregate nearest neighbor queries in road networks. *IEEE Trans. Knowl. Data Eng.* 17(6):820–833.

Zheng, B.; Zheng, K.; Xiao, X.; Su, H.; Yin, H.; Zhou, X.; and Li, G. 2016. Keyword-aware continuous knn query on road networks. In *ICDE*, 871–882.

Zhong, R.; Li, G.; Tan, K.-L.; Zhou, L.; and Gong, Z. 2015. G-tree: An efficient and scalable index for spatial search on road networks. *IEEE Trans. Knowl. Data Eng.* 27(8):2175–2189.

Zhu, L.; Jing, Y.; Sun, W.; Mao, D.; and Liu, P. 2010. Voronoi-based aggregate nearest neighbor query processing in road networks. In *GIS*, 518–521.