High-Throughput k Nearest Neighbors Search in Road Networks

YU KONG, The University of Sydney, Australia
LIJUN CHANG, The University of Sydney, Australia
DONG WEN, The University of New South Wales, Australia
DIAN OUYANG, Guangzhou University, China

Searching for the k nearest neighbors (kNN) among a given object set is a fundamental problem in road networks. In many real-world applications, such as location-based services, the object set is dynamic, with frequent insertions and deletions. In such workloads, *throughput* reflects the overall efficiency of an algorithm in handling both frequent queries and updates. The state-of-the-art solutions often suffer from slow query performance or inefficient index maintenance, which can result in low or even zero throughput. We propose a local subgraph-based indexing framework designed to support high-throughput query processing under frequent object updates. Unlike global indexing methods, our framework confines each update to a limited number of related local subgraphs, significantly reducing index maintenance overhead. To optimize throughput, we also introduce an efficient query processing strategy that incrementally explores only the necessary boundary vertices in relevant parts of our index, thereby pruning unnecessary computations. We formally prove that kNN results computed using only local subgraph information remain globally correct. Extensive experiments are conducted on 13 large real-world road networks to demonstrate the effectiveness and efficiency of our proposed solution.

CCS Concepts: • **Mathematics of computing → Graph algorithms**; • **Theory of computation → Shortest paths**; • **Information systems** → *Data management systems.*

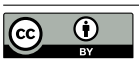
Additional Key Words and Phrases: kNN; road network; balanced tree hierarchy; hierarchical indexing

ACM Reference Format:
Yu Kong, Lijun Chang, Dong Wen, and Dian Ouyang. 2026. High-Throughput k Nearest Neighbors Search in Road Networks. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 42 (February 2026), 26 pages. https://doi.org/10.1145/3786656

1 Introduction

kNN search in road networks is a fundamental operation in many location-based services [24, 33, 36, 41]. Given a road network $G = (V, E)$, a set $\mathcal{M}$ of candidate objects, and a query vertex $q \in V$, the goal of kNN search is to identify the top-k objects in $\mathcal{M}$ closest to q based on shortest path distances. In a road network, vertices typically represent intersections and edges represent road segments, with edge weights modeling travel distance or time. Each object may represent a vehicle in a ride-hailing platform or an in-game entity in a location-based game, while a query vertex represents the user's or player's current location.

Authors' Contact Information: Yu Kong, The University of Sydney, Sydney, Australia, ykon0801@uni.sydney.edu.au; Lijun Chang, The University of Sydney, Sydney, Australia, lijun.chang@sydney.edu.au; Dong Wen, The University of New South Wales, Sydney, Australia, dong.wen@unsw.edu.au; Dian Ouyang, Guangzhou University, Guangzhou, China, dian.ouyang@gzhu.edu.cn.



ACM 2836-6573/2026/2-ART42
https://doi.org/10.1145/3786656

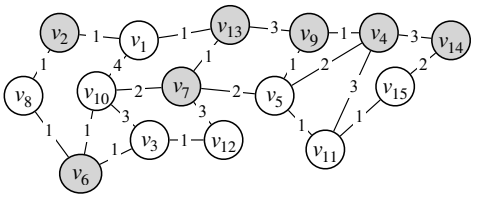


Fig. 1. A road network with objects on the gray vertices

Example 1.1. *Consider the road network in Fig. 1 with object set* $\mathcal{M} = \{v_2, v_4, v_6, v_7, v_9, v_{13}, v_{14}\}$. *For a query with* $k = 3$ *and* $q = v_9$, *the top-*3 *nearest neighbors of* q *in* $\mathcal{M}$ *are* $\{v_9, v_4, v_7\}$.

kNN search plays a central role in many real-world services. For example, in ride-hailing services such as *DiDi* [4], *Lyft* [7], and *Uber* [12], a basic operation is to efficiently identify a set of nearby available vehicles based on the pick-up location provided by a user. On accommodation booking platforms like *Booking* [2], *Airbnb* [1] and *Trip* [11], the system needs to present several available accommodations closest to the location specified by a user. In restaurant review services, such as *Yelp* [14], *Dianping* [3] and *OpenRice* [10], the providers are required to display a set of nearby restaurants relevant to the user's location. In location-based games such as *Pokémon* GO, objects represent in-game entities like *Pokémons*. The game server needs to find the *Pokémons* that are closest to a player. In modern delivery platforms such as *Meituan* [8], *Uber Eats* [13], and *DoorDash* [5], kNN search serves as a fundamental operation for identifying the nearest available riders to a given order location.

Two key system-level challenges arise in these applications.

(i) Dynamic Object Set: The object set $\mathcal{M}$ is frequently updated due to the mobility and availability of objects. For example, in ride-hailing platforms, vehicles continuously enter and exit the system as they become available or complete rides. In location-based games such as *Pokémon* Go, in-game entities like *Pokémons* appear and disappear dynamically across the map, leading to frequent updates. Similarly, in electric vehicle (EV) charging networks, charging stations may temporarily join or leave the available pool due to maintenance schedules, usage patterns, or availability. These scenarios result in a substantial *update load*, especially in large-scale deployments. For instance, *DiDi* has around 4 million active drivers [33], and each vehicle typically reports its location every 3 to 5 seconds [24]. *Meituan* hosts roughly 3.36 million monthly active riders [8, 9], with rider locations updated approximately every 10–30 seconds [6]. In the U.S., *Pokémon* Go's "nearby tracking" feature involved over 20 million daily players in 2016, each spending around 25 minutes in the game on average [33]. EV charging networks also exhibit frequent updates: Beijing targets 700,000 charging piles by 2025 [22], while India and the U.S. plan to expand from 75,000 to 375,000 and from 200,000 to over 500,000 by 2030 [25]. With ultra-fast charging reducing session durations to about 8 minutes [15, 25], station availability is updated increasingly often. Therefore, it is essential for the system to efficiently handle dynamic objects and support real-time kNN search.

(ii) High Throughput Requirement: Many existing solutions [36, 37, 41] focus on accelerating query processing. However, in dynamic environments with frequent updates, evaluating a kNN processing algorithm based solely on query latency offers only a *microscopic* perspective of system performance. Instead, *throughput* serves as a more *macroscopic* and comprehensive metric, capturing the system's ability to sustain real-time responsiveness under continuous load [24, 33]. Throughput is typically defined as the average number of queries the system can process per unit time in the presence of continuous updates [24, 33]. In high-demand applications such as *Didi* and *Pokémon* Go, limited throughput has been shown to cause service degradation [33] or even the removal of certain features [40], e.g., the "nearby tracking" function was disabled in *Pokémon* Go in July

2016 due to server overload [40]. Therefore, achieving high throughput is crucial for maintaining real-time responsiveness and overall system availability.

In this paper, we study the kNN search problem on road networks under dynamic object updates, aiming to optimize system throughput. Following [24, 35, 41], updates are defined as inserting or deleting objects from the object set $\mathcal{M} \subseteq V$.

Example 1.2. *Continuing Example 1.1, if v_7 is removed from the object set $\mathcal{M}$, then the kNN of $q = v_9$ is updated to $\{v_9, v_4, v_{13}\}$. Then, if a new object v_{11} is inserted, then $\mathcal{M}$ becomes $\{v_2, v_4, v_6, v_9, v_{11}, v_{13}, v_{14}\}$, and the kNN of q changes to $\{v_9, v_4, v_{11}\}$.*

Existing Solutions. The kNN search problem can be solved using Dijkstra's algorithm [20]. Given a query vertex q and an object set $\mathcal{M} \subseteq V$, the algorithm explores the graph by visiting vertices in non-decreasing order of their distances from q, terminating once k objects have been found. While object updates can be handled in constant time, query processing may become prohibitively slow on large road networks, especially when the nearest objects are far away from q. To address this challenge, numerous studies have proposed index structures aimed at improving query efficiency while maintaining acceptable update performance. These approaches can be broadly categorized into three groups: (1) accelerate Dijkstra's algorithm for kNN search using lightweight indexes, (2) extend distance labeling techniques for kNN search, and (3) fully materialize kNN answers. Representative methods in each category include GLAD [24], TEN-index [36], and KNN-index [41], respectively.

GLAD [24] constructs a grid-based index that partitions the road network into $2^x \times 2^x$ cells based on vertex coordinates and maintains, for each cell, a list of the objects it contains. While GLAD supports dynamic object updates efficiently, it suffers from relatively high query processing time. TEN-index [36] improves query efficiency by extending the H2H-index [35], a distance labeling technique, with additional auxiliary information to support kNN search. However, it incurs higher index maintenance overhead than GLAD, as more auxiliary data must be maintained. KNN-index [41], in contrast, fully materializes the kNN results for all vertices in a road network, enabling optimal query performance. However, as reported in [41], its average update efficiency is lower than that of TEN-Index, and its throughput is inferior in certain scenarios. Please refer to Section 3 for more details on these methods and other related work.

Present Work. In this paper, we propose a novel framework, CTLD-index, built on the core ideas of Cut tree and Local shortest distances to achieve a more balanced trade-off between query efficiency and update efficiency. Notably, our index outperforms TEN-index in both query performance and update cost. The key idea is as follows. Given a cut tree (e.g., the one constructed in [21]), which provides a hierarchical representation of vertex cuts in the road network G, we define a *partial vertex ordering*, denoted by $\prec$, meaning that some pairs of vertices are not comparable under $\prec$. For each vertex v, we define a ***local subgraph*** $G_{\succeq v}$ as the subgraph of G induced by vertices ranked lower than or equal to v according to this ordering. For each vertex v, our index stores the shortest distances computed within $G_{\succeq v}$ from v to all other vertices in the subgraph; we refer to these as ***local shortest distances***, in contrast to the *global* shortest distance computed over the entire network G. Based on these local distances, we also store the ***local k nearest neighbors*** of v, defined as the k closest objects within $G_{\succeq v}$ with respect to local shortest distances. We formally prove that the k nearest neighbors of any query vertex can be computed efficiently using only our index, without accessing the underlying road network. To further support efficient index construction and dynamic updates, we introduce the notion of *convex neighbor*. By confining index construction and maintenance to local subgraphs, our framework effectively localizes updates. This design achieves a favorable balance between low update overhead and high query efficiency, making it well-suited for a wide range of practical deployment scenarios.

Contributions. Our main contributions are summarized as follows.

- *A local subgraph-based kNN framework.* We propose a new indexing paradigm in which each vertex is associated with a local subgraph, and local kNN results are precomputed and stored based solely on local distance information within that subgraph. This eliminates the need for global structures and localizes index construction and maintenance, enabling efficient index maintenance under dynamic object updates.
- *Efficient algorithms.* We design an efficient query processing algorithm that incrementally traverses the index to retrieve accurate kNN results. We also develop efficient index construction algorithm and maintenance algorithms to handle object insertions and deletions with low overhead.
- *Theoretical analysis.* We formally prove that local subgraph-restricted kNN indexing yields correct results under general conditions. We also analyze the complexity bounds of our proposed algorithms.
- *Extensive experiments.* We evaluate our proposed algorithms on 13 large real-world road networks from different regions and compare them against state-of-the-art methods.

Outline. Section 2 defines the studied problem. Section 3 reviews related work. Section 4 introduces our index structure and query processing strategy. Section 5 presents the update algorithms for our proposed index. Section 6 introduces an improved index construction algorithm. Section 7 presents our experimental evaluation, and Section 8 concludes the paper. The source code and supplementary materials are available at https://github.com/ykon0801/CTLD-release.

2 Preliminaries

We consider a connected, undirected and weighted graph $G = (V, E)$ representing a road network, where V denotes the set of vertices and E the set of edges; each vertex carries an integer ID. We denote the number of vertices and edges in G by $n = |V|$ and $m = |E|$, respectively. For a vertex v in G, let $nbr(v) = \{u \in V \mid (v, u) \in E\}$ denote the set of v's neighbors in G. The degree of v is defined as $deg(v) = |nbr(v)|$. The weight of edge $(v, u) \in E$ is denoted by $\phi(v, u)$, and we assume that all weights are positive, i.e., $\phi(v, u) \in \mathbb{R}^+$. A path $P(v, u)$ in G is a sequence of vertices represented as $P = (v = v_1, v_2, \ldots, v_x = u)$, where $v_i \in V$ and $(v_i, v_{i+1}) \in E$ for each $1 \leq i < x$. The length of P is given by $\phi(P) = \sum_{i=1}^{x-1} \phi(v_i, v_{i+1})$. For any two vertices v and u in G, the shortest path between v and u is a path $P(v, u)$ that minimizes $\phi(P(v, u))$. The distance between v and u in G, denoted as $dist(v, u)$, is defined as the length of the shortest path between them, i.e., $dist(v, u) = \min\{\phi(P) \mid P \text{ is a path between } v \text{ and } u\}$. There is a set of moving objects associated with G, denoted by $\mathcal{M}$, and also a query point q. For presentation simplicity, we assume that both the objects and the query point are located on vertices, following previous works [24, 33, 36, 39, 41]. We will extend our techniques to directed graphs and to query points and objects located anywhere along the edges, and evaluate them in the experiments.

Definition 2.1 (kNN Query)**.** Given a road network $G = (V, E)$, an integer k, a query vertex q, and a set of candidate objects $\mathcal{M}$ with $|\mathcal{M}| > k$, a kNN query is to find the set of q's k nearest neighbors in $\mathcal{M}$, i.e., $\mathcal{M}_k(q) \subset \mathcal{M}$ such that:

(1) $|\mathcal{M}_k(q)| = k$; and
(2) $\forall v \in \mathcal{M}_k(q), v' \in \mathcal{M} \setminus \mathcal{M}_k(q), dist(q, v) \leq dist(q, v')$.

Following previous work [35], we assume that the shortest distance between every pair of vertices is unique. This assumption ensures that the result of any kNN search is also unique and thus simplifies our presentation. Nevertheless, our algorithms guarantee that the query result

Table 1. Frequently used notations

Notation	Description
$\mathcal{M}_k(q)$	top-k nearest neighbor set of q
T_G	cut tree of G
$\prec$	partial order over V defined by the cut tree T_G
$N(v)$	the tree node in T_G that contains vertex v
$\mathbb{S}(v)$	the set of vertices u s.t. $v \preceq u$
$\mathbb{P}(v)$	the set of vertices u s.t. $u \preceq v$
$G_{\succeq v}$	the local subgraph of v
$\mathcal{M}_k^{\succeq}(v)$	v's local k nearest neighbors in $G_{\succeq v}$
$lsd(v,u)$	local shortest distance from v to u computed in $G_{\succeq v}$
$dist(v,u)$	(global) shortest distance between v and u computed in G
$cn(v)$	the convex neighbors of v

Table 2. Parameters for throughput

Notation	Description	Values
T	duration of periods (in seconds)	1, **4**, 8, 16, 20, 24, 32
R^*	response requirement (ms)	0.2, 0.4, **0.8**, 1.6, 3.2, 6.4
$\rho = \frac{\lambda_u}{\hat{m}}$	object update rate (per second)	4, 2, 1, 0.5, **0.25**

$\mathcal{M}_k(q)$ for any query vertex q remains well-defined and valid even when the assumption does not hold. Frequently used notations are summarized in Table 1.

In real-world applications, the candidate object set $\mathcal{M}$ is typically dynamic, with frequent insertions and deletions. This work aims to support efficient kNN search under such dynamic conditions, with a focus on maximizing system throughput, as discussed next.

2.1 Throughput Model

We adopt the throughput model proposed by Luo et al. [33], which, to the best of our knowledge, is the only available throughput model for kNN search and has been widely used in subsequent studies [24, 36, 41]. It is important to note that this model does not assess performance under a specific workload; instead, it estimates the ***theoretical throughput***, i.e., the expected number of queries that can be processed per unit time. Let t_q denote the average query processing time (in seconds). In a query-only setting, the ideal throughput is therefore $\frac{1}{t_q}$; hence, a smaller t_q implies higher throughput. When dynamic updates are also involved, however, the analysis becomes more complex. Let t_u denote the average update processing time, and σ_q^2 and σ_u^2 be the corresponding variances of t_q and t_u. The model computes the overall throughput as a function of t_q, t_u, σ_q^2, and σ_u^2, along with some other system parameters (summarized in Table 2), under two distinct operational scenarios.

BUA+QF. In this scenario, time is divided into consecutive periods (or intervals), each lasting T seconds. This setup is referred to as batch update arrival (BUA). During each period, each object in $\mathcal{M}$ is updated exactly once, resulting in $\hat{m} = |\mathcal{M}|$ updates per period. Consequently, the total time required to process all updates within a period is $\hat{m} \cdot t_u$. The model also introduces a query response requirement, denoted as R^*, which specifies that the query response time must not exceed R^*. Queries are assumed to arrive according to a Poisson process. The query response time is then defined as the elapsed time from a query's arrival to its result delivery, and is equal to the query delay time plus t_q. Under the query-first (QF) policy, queries are given higher priority than updates, thereby minimizing query delay. Nevertheless, the system must still reserve sufficient time in each period to process all updates. As a result, only $T - \hat{m} \cdot t_u$ seconds are available for query processing

within a single period. Given this, the system can process at most $\frac{T-\hat{m}\cdot t_u}{t_q}$ queries per period, and the throughput is thus upper bounded by

$$\frac{1-\hat{m}/T\cdot t_u}{t_q} \tag{1}$$

Furthermore, if $\frac{\hat{m}t_u}{T}\cdot\frac{R^*}{t_q} < 0.5$, the query delay and update overhead are negligible, and the throughput can be approximated by $\frac{1}{t_q}$ [33].

RUA+FCFS. In this scenario, both queries and updates occur randomly (Random Update Arrival (RUA)). All operations are processed in the order of their arrival (First-Come-First-Served (FCFS)). The updates arrive according to a Poisson process with a rate of λ_u updates per second, which is typically set proportional to the object count $\hat{m}$, i.e., $\lambda_u = \rho\cdot\hat{m}$; thus, ρ controls the per-object update rate. As derived and proven in [33], the throughput is upper bounded by

$$\min\left\{\frac{2(R^*-t_q)(1-\lambda_u t_u)-\lambda_u(\sigma_u^2+t_u^2)}{\sigma_q^2+2R^*t_q-t_q^2},\ \frac{1-\lambda_u t_u}{t_q}\right\} \tag{2}$$

where σ_q^2 and σ_u^2 are the variances of query and update time.

3 Related Work

We categorize the related work into four groups. The first three are directly relevant to the problem of kNN search under dynamic updates, which is the focus of this paper. The fourth group includes studies that address related but distinct problems.

Accelerating Dijkstra's Algorithm for kNN Search Using Lightweight Indexes. As discussed in the Introduction, Dijkstra's algorithm [20] can solve the kNN search problem without any preprocessing or indexing; however, its query processing time is prohibitively high for online applications. IER [37] improves Dijkstra's search by leveraging the Euclidean distance between geographic coordinates (i.e., longitude and latitude) associated with vertices as a pruning bound to reduce unnecessary vertex exploration. While this offers some efficiency gains, it remains inadequate for real-time use. To address these limitations, several lightweight indexing techniques have been proposed to accelerate Dijkstra-based kNN search. ROAD [30] constructs a hierarchical partitioning of the road network and reduces the search space by skipping entire subgraphs that do not contain any objects. G-tree [44] and V-tree [39] adopt similar hierarchical structures but enhance performance by storing auxiliary summaries (e.g., nearest objects at border vertices) within each partition. GLAD [24], the state of the art in this category, introduces a grid-based index that partitions the road network into $2^x \times 2^x$ cells based on vertex coordinates, where x is a tunable parameter. Additionally, GLAD computes a shortest distance index—specifically the H2H-index [35]—to efficiently obtain exact distances between vertex pairs. All methods in this category support dynamic object updates efficiently due to their lightweight index structures. However, their query processing times are generally slower than those of more heavyweight indexing approaches discussed below. While GLAD offers the best query performance among lightweight methods, its reliance on the H2H-index leads to a large index size and substantial preprocessing time. We include GLAD as a representative of this category in our experiments.

Extending Distance Labeling Techniques for kNN Search. Another line of research extends distance labeling techniques to support kNN search. TOAIN [33] builds on contraction hierarchies [23] and stores partial k nearest neighbors for each vertex v by considering only objects with lower ranks than v. TEN-index [36], on the other hand, extends the H2H-index [35], which is based on tree decomposition [42]. For each vertex v, TEN-index stores partial k nearest neighbors among the

objects that are descendants of v in the tree decomposition. During query processing, both methods leverage these precomputed partial kNNs. However, TOAIN requires a Dijkstra-like search over a shortcut graph due to the nature of contraction hierarchies, whereas TEN-index avoids such search by exploiting the hierarchical structure of the tree decomposition. As a result, TEN-index achieves significantly better query performance than TOAIN and represents a state-of-the-art solution in this category, offering a strong balance between query performance and update efficiency. We include TEN-index in our experiments as a representative method from this category. Compared with the lightweight indexing techniques discussed earlier, methods in this category offer substantially faster query processing. However, they incur higher index maintenance overhead, as more auxiliary information must be maintained to support efficient queries.

Full Materialization of kNN Answers. Recently, KNN-index [41] was proposed to fully materialize the kNN search results for all vertices in a road network. This design enables optimal query performance, making it well-suited for workloads with frequent queries. KNN-index introduces an index structure that supports both efficient initial computation of all kNN results as well as their maintenance under dynamic object updates. However, despite these optimizations, KNN-index incurs high update costs due to the full materialization of kNN answers. This limitation is evident in the experimental results reported in [41], and our own experiments further confirm that KNN-index struggles to handle frequent updates within acceptable time bounds.

Other Related Work. There are also other relevant studies that tackle different problem settings and are orthogonal to ours. TD-H2H [32] extends GLAD [24] and H2H-index [35] to support nearest neighbor search in time-dependent road networks, where edge costs vary over time. Continuous nearest neighbor (CNN) queries, where the objects remain static but the query vertex itself moves along the road network, have also been extensively studied [17–19, 26–29, 34, 38, 43]. In addition, GPU parallelism was exploited to accelerate kNN query processing in [31].

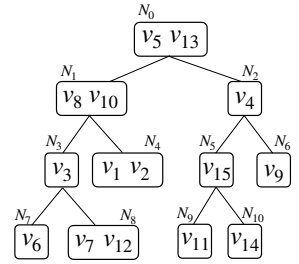
4 Our Approach

As shown in Section 2.1, the throughput is upper bounded by $\frac{1-\hat{m}/T \cdot t_u}{t_q}$ (in BUA+QF) or $\frac{1-\lambda_u t_u}{t_q}$ (in RUA+FCFS), where t_q and t_u denote the average query and update time, and both $\frac{\hat{m}}{T}$ and λ_u represent the average number of updates per second. Thus, achieving higher throughput requires minimizing both t_q and t_u. In practice, however, there is an inherent trade-off between query and update efficiency: optimizing for faster queries often leads to slower updates [33]. For instance, if the exact kNN results are precomputed and stored for every vertex (as in KNN-index [41]), the query time t_q is minimized, but the update time t_u increases dramatically, potentially resulting in zero throughput when $t_u \geq \frac{1}{\lambda_u}$. Motivated by this observation, we in this paper propose a new index structure for dynamic kNN search that achieves efficient updates while supporting queries more efficiently than all existing approaches except KNN-index. We first introduce the index structure and its construction algorithm in Section 4.1, and then present index-based query processing algorithm in Section 4.2; index maintenance techniques will be discussed in Section 5.

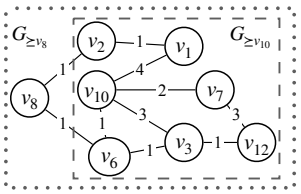
4.1 Our CTLD-Index

Our CTLD-index is designed based on the core ideas of Cut tree and Local shortest distances. Cut tree was originally introduced in [21] for shortest distance indexing and querying, and is defined via recursively finding vertex cuts.

Definition 4.1 (Vertex Cut)**.** Given a graph $G = (V, E)$, a vertex subset $C \subset V$ is a vertex cut if removing them disconnects the graph, i.e., $G \setminus C$ consists of at least two connected components.



(a) Cut tree T_G



(b) Two local subgraphs

	lsd	$\mathcal{M}_k^{\succeq}(\cdot)$
v_1	0, 4, 2, 1, 4	v_2
v_2	0, 1, 5, 1, 2, 5	v_2
v_3	0, 2, 2, 5, 6	v_6, v_7
v_4	0, 4, 2	v_4, v_9, v_{14}
v_5	0	v_9, v_4, v_7
v_6	0, 1, 1, 1, 4, 5	v_6
v_7	0, 4, 2, 4, 1, 2	v_7
v_8	0, 3, 6	v_2, v_6, v_7
v_9	0, 1, 3, 1	v_9
v_{10}	0, 2, 3, 4	v_6, v_7, v_2
v_{11}	0, 1, 3, 7, 1	$\emptyset$
v_{12}	0, 3, 1, 3, 3, 4, 5	$\emptyset$
v_{13}	0, 3	v_{13}, v_7, v_2
v_{14}	0, 2, 3, 7, 4	v_{14}
v_{15}	0, 4, 8, 2	v_{14}

(c) Labels ($k = 3$)

Fig. 2. Our index constructed for the graph in Fig. 1

It is straightforward that, for any two vertices v and u that are in different connected components of $G \setminus C$, every path between them must have at least one vertex in C. As a result,

$$dist(v, u) = \min_{c \in C} dist(v, c) + dist(c, u) \tag{3}$$

The cut tree constructs a hierarchical representation of vertex cuts.

Definition 4.2 (Cut Tree)**.** Given a graph $G = (V, E)$, a cut tree of G, denoted as T_G, is a rooted **binary** tree such that

(1) Each node $N \in T_G$ is a **disjoint** subset of V.
(2) The nodes of T_G cover all vertices of G, i.e., $V = \bigcup_{N \in T_G} N$.
(3) Each node $N \in T_G$ is a vertex cut of the subgraph g of G induced by all vertices in the subtree rooted at N. Moreover, vertices in the subtree rooted at N's left (resp. right) child correspond to unions of connected components of $g \setminus N$.

To avoid ambiguity, we use the term "nodes" to refer to nodes of the cut tree and "vertices" to refer to vertices of a graph throughout the remainder of the paper. For a vertex $v \in V$, we use $N(v)$ to denote the cut tree node that contains v.

Example 4.1. *Consider the graph G shown in Fig. 1. $C = \{v_5, v_{13}\}$ is a vertex cut of G, as removing these vertices splits G into two disconnected components. Fig. 2a shows the entire cut tree T_G. Vertices in node N_1 (i.e., $\{v_8, v_{10}\}$) form a cut of the subgraph of G induced by vertices in the subtree rooted at N_1 (i.e., $\{v_1, v_2, v_3, v_6, v_7, v_8, v_{10}, v_{12}\}$).*

A cut tree T_G of G induces a partial vertex ordering of V.

Definition 4.3 (Partial Vertex Ordering)**.** For any two vertices $v, u \in V$, we say that v ranks higher than u with respect to T_G, denoted as $v \prec u$, if one of the following two conditions holds:

(1) v and u are in different tree nodes (i.e., $N(v) \neq N(u)$), and $N(v)$ is an ancestor of $N(u)$ in T_G.
(2) $N(v) = N(u)$ and the ID of v is smaller than the ID of u.

Note that in the partial vertex ordering, when v and u are in the same tree node, their ordering could be arbitrary. For concreteness, we order them based on their IDs.

Based on the partial vertex ordering $\prec$, we define two key sets for each vertex $v \in V$. Specifically, let $\mathbb{S}(v) = \{v\} \cup \{u \in V \mid v \prec u\}$ be the successor set of v, i.e., the union of v and the set of vertices that rank lower than v, and $\mathbb{P}(v) = \{v\} \cup \{u \in V \mid u \prec v\}$ be the predecessor set of v, i.e., the union of v and the set of vertices that rank higher than v. Note that, $v \in \mathbb{S}(v)$ and $v \in \mathbb{P}(v)$. We also remark that ***although $\prec$ is a partial vertex ordering, it induces a total ordering for vertices of*** $\mathbb{P}(v)$; this is because for any two vertices $u, u' \in \mathbb{P}(v)$, either $N(u) = N(u')$ or one of $N(u)$ and $N(u')$ is an ancestor of the other. Given a cut tree T_G, the partial vertex ordering $\prec$ and the sets $\mathbb{S}(\cdot)$ and $\mathbb{P}(\cdot)$ are uniquely defined. We then define local subgraph and local shortest distance based on $\mathbb{S}(\cdot)$, which are crucial to our index.

Definition 4.4 (Local Subgraph and Local Shortest Distance)**.** Given a cut tree T_G of G and a vertex $v \in V$, the local subgraph of v, denoted as $G_{\succeq v}$, is the subgraph of G induced by $\mathbb{S}(v)$. The local shortest distance between v and any vertex u in $G_{\succeq v}$, denoted as $lsd(v, u)$, is defined as their shortest distance computed in $G_{\succeq v}$.

Example 4.2. *Consider the cut tree in Fig. 2a. We have $v_5 \prec v_{13} \prec v_8 \prec v_{10}$, while v_4 and v_8 are incomparable regarding $\prec$. Also, $\mathbb{P}(v_5) = \{v_5\}$ and $\mathbb{S}(v_5)$ includes all vertices of the graph G; similarly, $\mathbb{P}(v_8) = \{v_5, v_8, v_{13}\}$, $\mathbb{S}(v_8) = \{v_1, v_2, v_3, v_6, v_7, v_8, v_{10}, v_{12}\}$, and $\mathbb{S}(v_{10}) = \mathbb{S}(v_8) \setminus \{v_8\}$. The local subgraphs $G_{\succeq v_8}$ and $G_{\succeq v_{10}}$ are shown in Fig. 2b, highlighted with dotted and dashed lines, respectively. The shortest distance between v_{10} and v_2 in G is 3, while $lsd(v_{10}, v_2) = 5$.*

Note that $lsd(v, v) = 0$ and $lsd(\cdot, \cdot)$ is ***asymmetric***. For any two vertices v and u such that $v \neq u$, at most one of $lsd(v, u)$ and $lsd(u, v)$ is defined; specifically, $lsd(v, u)$ is defined if and only if $v \prec u$. Moreover, when defined, $lsd(v, u)$ exists only within the local subgraph $G_{\succeq v}$ and not in any other subgraphs. For example, $lsd(v_{10}, v_2)$ is defined only within $G_{\succeq v_{10}}$.

Index Structure. Based on the above definitions, for a given road network $G = (V, E)$, an integer k and a set of candidate objects $\mathcal{M}$, our CTLD-index consists of the following components.

(1) A cut tree T_G of G.
(2) For each vertex $v \in V$, the local shortest distances between v and all vertices of $G_{\succeq v}$, i.e., $\big\{\big(v, u, lsd(v, u)\big) \mid u \in \mathbb{S}(v)\big\}$.
(3) For each vertex $v \in V$, the local k nearest neighbors of v computed in $G_{\succeq v}$, denoted as $\mathcal{M}_k^{\succeq}(v)$. Note that $\mathcal{M}_k^{\succeq}(v) \subseteq \mathcal{M} \cap \mathbb{S}(v)$, and the objects of $\mathcal{M} \cap \mathbb{S}(v)$ are compared with each other based on their local shortest distances to v.

Example 4.3. *Consider the graph in Fig. 1 and its cut tree in Fig. 2a. Given that $\mathcal{M} = \{v_2, v_4, v_6, v_7, v_9, v_{13}, v_{14}\}$ and $k = 3$, Fig. 2c summarizes the distance labels (2nd column) and $\mathcal{M}_k^{\succeq}(\cdot)$ sets (3rd column) for each vertex. Note that, **for easy access, we store $lsd(v, u)$ at vertex u, instead of** v. As a result, the local shortest distances stored at vertex v_8 are $lsd(v_8, v_8) = 0$, $lsd(v_{13}, v_8) = 3$ and $lsd(v_5, v_8) = 6$; moreover, we do not need to explicitly store vertices associated with these distances, since they can be inferred from v_8's predecessors in ranked order, i.e., $v_5 \prec v_{13} \prec v_8$. Additionally, v_8 records $\mathcal{M}_k^{\succeq}(v_8)$ as $\{v_2, v_6, v_7\}$, with corresponding distances $lsd(v_8, v_2) = 1$, $lsd(v_8, v_6) = 1$, and $lsd(v_8, v_7) = 4$.*

Note that the index is built for a predefined k and supports queries for any $k' \leq k$. In real-world applications such as ride-hailing, kNN queries typically involve a small k relative to $|V|$ (e.g., retrieving a few nearby drivers rather than all drivers). Therefore, when constructing the index, k is usually set to a moderate value [24, 33, 36, 41]. Also, the above description of our CTLD-index includes only the three core components required for processing kNN queries. To support efficient updates, our CTLD-index includes an additional component, which will be introduced in Section 5.2.

As $\mathcal{M}_k^{\succeq}(v) \subseteq \mathbb{S}(v)$, the size of our index is bounded as follows.

Theorem 4.1. *The size of our* CTLD*-index is $O(\sum_{v \in V} |\mathbb{S}(v)|) = O(\sum_{v \in V} |\mathbb{P}(v)|)$.*

Algorithm 1: CTLD-Cons$(g, k, \mathcal{M})$

Input: A graph g, an integer k and a set of candidate objects $\mathcal{M}$
Output: A cut tree T_g of g, and associated local shortest distances and local k nearest neighbors

1 $(L, C, R) \leftarrow$ BalancedCut(g);
2 **foreach** $c \in C$ *in increasing ID order* **do**
3 Run Dijkstra's algorithm in g, starting from c, to compute local shortest distances $\{(c, v, lsd(c, v)) \mid v \in V(g)\}$ and local k nearest neighbors $\mathcal{M}_k^{\succeq}(c)$;
4 Remove c from g;
5 **if** $|L| > 2$ **then** $T_{g[L]} \leftarrow$ CTLD-Cons$(g[L], k, \mathcal{M})$;
6 **else** $T_{g[L]} \leftarrow L$;
7 **if** $|R| > 2$ **then** $T_{g[R]} \leftarrow$ CTLD-Cons$(g[R], k, \mathcal{M})$;
8 **else** $T_{g[R]} \leftarrow R$;
9 Create a node for C and set it as the parent of $T_{g[L]}$ and $T_{g[R]}$ in T_g;
10 **return** T_g

Index Construction Algorithm. Our index construction algorithm is a recursive algorithm, as shown in Algorithm 1. Each recursion consists of three phases. The first phase identifies a vertex cut C of the given graph g (Line 1). The second phase computes the local shortest distances between each $c \in C$ and all vertices $v \in \mathbb{S}(c)$, as well as local k-nearest neighbors $\mathcal{M}_k^{\succeq}(c)$ (Lines 2–4). The third phase (Lines 5–9) conducts recursion for $g[L]$ and $g[R]$, the subgraphs of g induced by vertex subsets L and R.

We adopt the techniques of [21] for computing the vertex cut in Line 1. The computed vertex cut is not an arbitrary one, but is a balanced cut with a balancing parameter β; we use the default $\beta = 0.2$ in our implementation. Let C be a vertex cut of g with L and R being the resulting partitions that are not adjacent to each other in $g \setminus C$; note that, L and R themselves could be disconnected. (L, C, R) is a balanced cut with respect to β if $\max\{|L|, |R|\} \leq (1 - \beta)|V(g)|$. The general idea of computing a balanced cut is as follows; please refer to [21] for details. For a connected graph g, the procedure begins by selecting two vertices v_L and v_R as far apart as possible, and generating two initial partitions based on the vertices' distances to v_L and v_R, each containing approximately $\beta \cdot |V(g)|$ vertices, with the remaining vertices forming the cut region. Next, each initial partition is contracted into a supernode, and a minimal cut is identified to separate the two supernodes in the resulting graph. Finally, each connected component in the remaining cut region is assigned to one of the two initial partitions, aiming to maximize balance; this produces the balanced cut (L, C, R). For a disconnected graph g, if the largest connected component of g exceeds $(1-\beta)|V(g)|$, a balanced cut is computed within this component following the same procedure as above, and the remaining connected components of g are then assigned to the two resulting partitions aiming to maximize balance. Otherwise, BalancedCut assigns connected components of g to two partitions while maximizing balance, and the separating cut C is empty. Note that an empty cut would not arise if non-binary cut trees were allowed; however, the resulting index structure remains similar. In this paper, we maintain a strictly binary tree structure. As shown in [21], the time complexity of BalancedCut is $O\big(|E(g)| \cdot (\log |V(g)| + |C|)\big)$.

To compute local shortest distances and local k nearest neighbors in Lines 2–4, we process the vertices c of C in increasing ID order, and remove c from g once it is processed. As a result, when processing c, $\mathbb{S}(c)$ is always the entire set of vertices in the current graph, and the local shortest distances and the local k nearest neighbors of c can all be computed at the same time by a single instance of Dijkstra's algorithm [20] with c as the source vertex.

Theorem 4.2. *The time complexity of our index construction (i.e.,* CTLD-*Cons*$(G, k, \mathcal{M})$*) is* $O((m \log n + w(m + n \log n)) \log n)$*, where* w *is the largest number of vertices contained in any node of* T_G*.*

Remarks. Although we adopt the techniques of [21] for computing the vertex cut in Line 1 of Algorithm 1, our cut tree is different from and is structurally more efficient than the one constructed by [21]. This is because the vertex cuts computed in [21] are ***global*** shortest distance preserving. That is, after obtaining the cut (L, C, R) by BalancedCut, they will construct two subgraphs, g_L for L and g_R for R, such that the shortest distance between any two vertices computed in g_L (resp. g_R) is the same as the global distance computed in g. To achieve this, they need to add shortcuts to $g[L]$ and $g[R]$ to retain the global shortest distances. In contrast, we simply pass $g[L]$ and $g[R]$ to the subroutines in Algorithm 1. Our design offers two advantages. First, it eliminates the need to compute and add shortcut edges to preserve global distances in subgraphs, thereby simplifying the indexing process and improving the construction time. Second, $g[L]$ (resp. $g[R]$) is typically sparser than g_L (resp. g_R), and thus our cut tree is likely to be more compact. We will empirically evaluate these advantages of our design in Section 7. Despite these simplifications, we will show in Section 4.2 that local shortest distances and local k nearest neighbors computed based on our cut tree are sufficient for processing any kNN queries.

Although our index also stores partial k nearest neighbors for each vertex, similar to TOAIN and TEN-index, it differs from them in several key aspects. First, while TOAIN and TEN-index compute partial kNNs based on contraction hierarchies and tree decomposition, respectively, our method is based on a cut tree structure. Secondly, TOAIN and TEN-index rely on global distances computed over the entire graph G, whereas we compute partial kNNs using local distances within localized subgraphs. Thanks to this localized computation, our index achieves lower construction and update time, as well as faster query processing compared to TEN-index, as demonstrated in our experiments. To the best of our knowledge, we are the first to leverage local distances for kNN search indexing.

4.2 Index-based Query Processing

In this subsection, we propose an algorithm to efficiently find the k nearest neighbors for any query vertex q, based on our CTLD-index constructed in Section 4.1. Before that, we first prove important properties of our index in the following lemmas and theorems.

Lemma 4.1. *For any path* P *in* G*, there exists a vertex* $c \in P$ *such that* P *is contained in* $G_{\succeq c}$*. Specifically, let* v, u *be* P*'s two end-points. The highest-ranked vertex in* $P \cap \mathbb{P}(v) \cap \mathbb{P}(u)$ *is such a vertex* c*.*

The general idea is as follows. Let c^* be the highest-ranked vertex in $P \cap \mathbb{P}(v)$, which is well-defined since (1) $P \cap \mathbb{P}(v)$ is non-empty (because $v \in P \cap \mathbb{P}(v)$) and (2) the vertices of $\mathbb{P}(v)$ are totally ordered. One can show that P is entirely contained within $G_{\succeq c^*}$. Consequently, c^* is also the highest-ranked vertex in $P \cap \mathbb{P}(u)$.

Then, the shortest distance between any two vertices v and u in G can be computed from our CTLD-index, as follows.

Theorem 4.3 (**Shortest Distance Computation**). *For any two vertices* $v, u \in V$*, their shortest distance in* G *can be computed from our* CTLD*-index as,*

$$dist(v, u) = \min_{c \in \mathbb{P}(v) \cap \mathbb{P}(u)} lsd(c, v) + lsd(c, u) \tag{4}$$

where $lsd(c, v)$ *and* $lsd(c, u)$ *are stored in our index.*

Furthermore, the k nearest neighbors for any query vertex q can be computed based on the theorem below.

Theorem 4.4 (**kNN Computation**). *For any query vertex q, its k nearest neighbors are the top-k objects with the smallest distances in*

$$C = \bigcup_{c \in \mathbb{P}(q)} \left\{ \left(o, lsd(c, o) + lsd(c, q) \right) \mid o \in \mathcal{M}_k^{\succeq}(c) \right\} \tag{5}$$

after removing duplicate objects and preferring shorter distances.

Query Processing Algorithm. A naive algorithm for query processing would directly follow Theorem 4.4, i.e., first obtaining C by Equation (5), and then retrieving the top-k objects from C using a heap of size k. The time complexity of this naive algorithm is $O(|\mathbb{P}(q)| \cdot k \cdot \log k)$, since it always examines all entries of C which is of cardinality $|C| = O(|\mathbb{P}(q)| \cdot k)$.

Algorithm 2: CTLD-Query

Input: Our CTLD-index, a query vertex q and an integer k
Output: The set $\mathcal{M}_k(q)$ of k nearest neighbors of q

1 $\mathcal{M}_k(q) \leftarrow \emptyset$;
2 $\mathcal{Q} \leftarrow$ an empty min-priority queue;
3 **foreach** $c \in \mathbb{P}(q)$ **do**
4 **if** $\mathcal{M}_k^{\succeq}(c) = \emptyset$ **then continue**;
5 $o \leftarrow$ the first object in $\mathcal{M}_k^{\succeq}(c)$;
6 Push the entry $(lsd(c, q) + lsd(c, o), c, o, 1)$ into $\mathcal{Q}$;
7 **while** $|\mathcal{M}_k(q)| < k$ **and** $\mathcal{Q} \neq \emptyset$ **do**
8 $(d, c, o, idx) \leftarrow$ pop the top entry from $\mathcal{Q}$;
9 **if** $o \notin \mathcal{M}_k(q)$ **then** $\mathcal{M}_k(q) \leftarrow \mathcal{M}_k(q) \cup \{o\}$;
10 **if** $|\mathcal{M}_k(q)| < k$ **and** $idx < |\mathcal{M}_k^{\succeq}(c)|$ **then**
11 $o' \leftarrow$ the $(idx + 1)$-th object in $\mathcal{M}_k^{\succeq}(c)$;
12 Push the entry $(lsd(c, q) + lsd(c, o'), c, o', idx + 1)$ into $\mathcal{Q}$;
13 **return** $\mathcal{M}_k(q)$;

To reduce unnecessary evaluations, we propose an improved query processing algorithm which may only need to examine a small number of entries of $\mathcal{M}_k^{\succeq}(c)$ for each $c \in \mathbb{P}(q)$. Assuming entries of $\mathcal{M}_k^{\succeq}(c)$ are in ascending order regarding $lsd(c, \cdot)$, the general idea is that: if the i-th object of $\mathcal{M}_k^{\succeq}(c)$ is not in the final result $\mathcal{M}_k(q)$, then any lower ranked object (i.e., $(i+1)$-th, $(i+2)$-th, ..., k-th objects) of $\mathcal{M}_k^{\succeq}(c)$ will not be in $\mathcal{M}_k(q)$ and thus do not need to be examined. The pseudocode of our algorithm is described in Algorithm 2. We use a min-priority queue $\mathcal{Q}$ to store the frontier (i.e., the first object to be considered) of $\mathcal{M}_k^{\succeq}(c)$ for each $c \in \mathbb{P}(q)$ (Lines 2–6), where the ranking key in $\mathcal{Q}$ is the distance d. While the k nearest neighbor of q has not been identified and $\mathcal{Q}$ is not empty (Line 7), we pop the top entry from $\mathcal{Q}$ (Line 8), which includes an object o and the corresponding $c \in \mathbb{P}(q)$ such that o is the idx-th object of $\mathcal{M}_k^{\succeq}(c)$. If o is not already in $\mathcal{M}_k(q)$, it is the next nearest neighbor of q (Line 9). We also move the frontier of $\mathcal{M}_k^{\succeq}(c)$ to the next object, and push it into $\mathcal{Q}$ (Lines 10–12).

Example 4.4. *Consider the index in Fig. 2, a query vertex $q = v_{10}$ and $k = 3$. We first obtain $\mathbb{P}(q) = \{v_5, v_8, v_{10}, v_{13}\}$. After the "for" loop in Lines 3–6, $\mathcal{Q}$ contains four entries, one from each $c \in \mathbb{P}(q)$: $\left(lsd(v_5, v_{10}) + lsd(v_5, v_9), v_5, v_9, 1\right) = (1 + 4, v_5, v_9, 1)$, $(2 + 1, v_8, v_2, 1)$, $(0 + 1, v_{10}, v_6, 1)$, and $(3 + 0, v_{13}, v_{13}, 1)$. In the first iteration of the "while" loop in Line 7, we pop the top entry $(1, v_{10}, v_6, 1)$ from $\mathcal{Q}$. As $v_6 \notin \mathcal{M}_k(q)$, we add v_6 as the first nearest neighbor to $\mathcal{M}_k(q)$. We then push the next object v_7 of $\mathcal{M}_k^{\succeq}(v_{10})$, represented as $(0 + 2, v_{10}, v_7, 2)$, into $\mathcal{Q}$. In the second iteration, the entry $(2, v_{10}, v_7, 2)$*

is popped from Q, v_7 is added as the second nearest neighbor to $\mathcal{M}_k(q)$, and $(0+5, v_{10}, v_2, 3)$ is pushed into Q.

We remark that Algorithm 2 supports early stopping to avoid unnecessary evaluations. Specifically, the algorithm terminates as soon as k objects have been inserted into $\mathcal{M}_k(q)$ (Line 7). Since the objects in $\mathcal{M}_k^{\succeq}(c)$ for each $c \in \mathbb{P}(q)$ are accessed sequentially in ascending order of their local distances (Line 11), the algorithm typically needs to scan only a small prefix of each $\mathcal{M}_k^{\succeq}(c)$. Let l_i^c denote the number of accessed objects of $\mathcal{M}_k^{\succeq}(c)$ when the i-th nearest neighbor of q is found. Then, l_i^c is bounded above by i, and is often much smaller than i in practice.

Theorem 4.5. *The time complexity of Algorithm 2 for finding the first i nearest neighbors is $O(\sum_{c\in\mathbb{P}(q)} l_i^c \cdot \log|\mathbb{P}(q)|)$.*

As $l_i^c \le i$, the time complexity is also bounded by $O(|\mathbb{P}(q)| \cdot i \cdot \log|\mathbb{P}(q)|)$, which is polynomial in i. Thus, Algorithm 2 has an *incremental polynomial* time complexity [16]. We remark that although $\sum_{c\in\mathbb{P}(q)} l_k^c$ can reach $|\mathbb{P}(q)| \cdot k$ and thus the time complexity of Algorithm 2 can reach $O(|\mathbb{P}(q)| \cdot k \cdot \log|\mathbb{P}(q)|)$ in the worst case, our empirical studies in Section 7 show that Algorithm 2 finds the k nearest neighbors by examining only a small portion of C.

5 Online Index Updating

In this section, we introduce techniques for efficiently updating our CTLD-index when objects are dynamically inserted or removed; object movements can be simulated as a removal followed by an insertion. Note that we consider the online updating (in contrast to batch updating) setting in this paper, i.e., the index is updated after each object insertion or deletion. Since the graph structure and edge weights remain fixed, both the cut tree and the precomputed local shortest distances remain unaffected. Consequently, only the local k nearest neighbors, $\mathcal{M}_k^{\succeq}(\cdot)$, require updating.

We propose algorithms to efficiently update $\mathcal{M}_k^{\succeq}(\cdot)$ for object deletion and insertion in Sections 5.3 and 5.4, respectively. We first describe a naive baseline in Section 5.1, and then introduce the concept of convex neighbor in Section 5.2 which plays a key role in the efficiency of our index updating.

5.1 A Naive Approach

Let o be the updated object, which could be either an insertion or a deletion. Then, only vertices of $\mathbb{P}(o)$ may have their local k nearest neighbors $\mathcal{M}_k^{\succeq}(\cdot)$ changed, as $o \notin \mathbb{S}(v)$ for any other vertex v. A straightforward approach is to consider all vertices of $\mathbb{P}(o)$, and for each such vertex $c \in \mathbb{P}(o)$, recompute $\mathcal{M}_k^{\succeq}(c)$ as the top-k objects in the updated $\mathcal{M}^{\succeq}(c)$, where $\mathcal{M}^{\succeq}(c)$ denotes the set of objects that are in the local subgraph $G_{\succeq c}$ (i.e., $\mathcal{M}^{\succeq}(c) = \mathcal{M} \cap \mathbb{S}(c)$). This recomputation is generally cheaper than running Line 3 of Algorithm 1, because the local shortest distances from c to all vertices of $\mathbb{S}(c)$ and thus all objects of $\mathcal{M}^{\succeq}(c)$ have already been precomputed and stored in our CTLD-index. As a result, we can simply scan all objects in $\mathcal{M}^{\succeq}(c)$ and retain the top-k objects with respect to their local shortest distances from c. However, it is still inefficient, as the number of objects in $\mathcal{M}^{\succeq}(c)$ can be large.

5.2 Computing $\mathcal{M}_k^{\succeq}(\cdot)$ via Convex Neighbors

To enable efficient updating $\mathcal{M}_k^{\succeq}(\cdot)$, an intuitive idea is to infer $\mathcal{M}_k^{\succeq}(c)$ from the local k nearest neighbors of c's lower-ranked neighbors in $G_{\succeq c}$. This is appealing, as one might expect that the local nearest neighbors of the nearby vertices could help reconstruct $\mathcal{M}_k^{\succeq}(c)$. However, this does not work in general, because $\mathcal{M}_k^{\succeq}(c) \setminus \{c\}$ is not guaranteed to be a subset of $\bigcup_{v\,:\, v\in nbr(c)\wedge c\prec v} \mathcal{M}_k^{\succeq}(v)$. For example, consider the index in Fig. 2 and the local subgraph $G_{\succeq v_8}$ in Fig. 2b. v_8's lower-ranked

neighbors are v_2 and v_6. $\mathcal{M}_k^{\succeq}(v_2) = \{v_2\}$, $\mathcal{M}_k^{\succeq}(v_6) = \{v_6\}$, while $\mathcal{M}_k^{\succeq}(v_8) = \{v_2, v_6, v_7\}$. Thus, using only neighbors' $\mathcal{M}_k^{\succeq}(\cdot)$ sets is insufficient to guarantee correctness.

In this subsection, we introduce the concept of convex neighbor and show that $\mathcal{M}_k^{\succeq}(c) \setminus \{c\}$ can be obtained from the local k nearest neighbors of c's convex neighbors. This enables us to efficiently update $\mathcal{M}_k^{\succeq}(\cdot)$. Convex neighbor is defined based on convex path.

Definition 5.1 (Convex Path)**.** A path $P(v, u) = (v, x_1, \ldots, x_j, u)$ is a convex path if each intermediate vertex ranks lower than both terminal vertices, i.e., $v \prec x_i$ and $u \prec x_i$ for each $1 \leq i \leq j$.

Note that for a path $P(v, u)$ to be a convex path, it must be contained within the local subgraph of the higher-ranked vertex between v and u. For example, assuming $v \prec u$, any convex path between v and u must be contained in $G_{\succeq v}$.

Definition 5.2 (Convex Neighbor)**.** Given two vertices v and u such that $v \prec u$, u is a convex neighbor of v if there exists a convex path between them in the local subgraph $G_{\succeq_v}$.

Denote the set of v's convex neighbors by $cn(v)$. Note that a vertex is not considered a convex neighbor of itself, i.e., $v \notin cn(v)$. Our index updating algorithm in Sections 5.3 and also our improved index construction algorithm in Section 6 are based on the theorem below, which is adapting Theorem 4.4 to local kNN computation.

Theorem 5.1 (**Local kNN Computation**). *For any vertex $c \in V$ such that $c \notin \mathcal{M}$, its local k nearest neighbors are the top-k objects with the smallest distances in*

$$C_L = \bigcup_{v \in cn(c)} \left\{ \left(o, lsd(c, v) + lsd(v, o)\right) \mid o \in \mathcal{M}_k^{\succeq}(v) \right\} \tag{6}$$

after removing duplicate objects and preferring shorter distances.

Note that, if $c \in \mathcal{M}$, then $\mathcal{M}_k^{\succeq}(c) \setminus \{c\}$ is the top-$(k-1)$ objects with the smallest distances in C_L. Following Theorem 5.1, we also precompute and store ***the convex neighbors of all vertices in*** V, which ***constitute the fourth component of our*** CTLD***-index***.

Example 5.1. *Reconsider the example given at the beginning of this subsection. The convex neighbors of v_8 are $cn(v_8) = \{v_1, v_2, v_3, v_6, v_{10}\}$. We have $\mathcal{M}_k^{\succeq}(v_8) \subseteq \bigcup_{v \in cn(v_8)} \mathcal{M}_k^{\succeq}(v)$ since $\mathcal{M}_k^{\succeq}(v_3) = \{v_6, v_7\}$.*

Convex Neighbor Computation. Now, we present an algorithm to compute the convex neighbors. First, we prove an important property of convex neighbor in the following lemma.

Lemma 5.1. *Let $v \prec u$. u is a convex neighbor of v if and only if either u is neighbor of v in $G_{\succeq v}$ or there exists another vertex x such that $u \prec x$ and x is a convex neighbor of both v and u.*

Lemma 5.1 implies that the convex neighbor relationships can be inferred incrementally through intermediate vertices of lower ranks. However, it remains nontrivial to directly utilize this property for efficiently computing convex neighbors. To achieve that, we introduce the concept of convex predecessor, the inverse of convex neighbor. Specifically, v is a ***convex predecessor*** of u if u is a convex neighbor of v. Denote the set of convex predecessors of a vertex v by $cp(v)$. Convex predecessor makes the relationship easier to be utilized, as shown in the corollary below.

Corollary 5.1. *Let $v \prec u$. v is a convex predecessor of u if and only if either u is a neighbor of v in $G_{\succeq v}$ or there exists another vertex x such that $u \prec x$ and both v and u are convex predecessors of x.*

Based on this insight, we propose a bottom-up approach (i.e., in reverse order of $\prec$) to efficiently computing convex predecessors and convex neighbors. The pseudocode is shown in Algorithm 3. For each vertex $u \in V$, we initialize its convex neighbor as empty (Line 2) and set its initial convex

Algorithm 3: ConvexNeighbor-Cons

Input: A graph $G = (V, E)$ and its cut tree T_G
Output: Convex neighbors $cn(v)$ for each $v \in V$

1 **foreach** $u \in V$ **do**
2 $\quad cn(u) \leftarrow \emptyset$;
3 $\quad cp(u) \leftarrow \{v \in V \mid v \in nbr(u) \wedge v \prec u\}$;
4 $Q \leftarrow$ an empty first-in-first-out queue;
5 **foreach** $v \in V$ *s.t.* $|\mathbb{S}(v)| = 1$ **do** Push v into Q;
6 **while** $Q \neq \emptyset$ **do**
7 $\quad x \leftarrow$ pop a vertex from Q;
8 $\quad$ **foreach** $v, u \in cp(x)$ *s.t.* $v \prec u$ **do** $cp(u) \leftarrow cp(u) \cup \{v\}$;
9 $\quad$ **foreach** $v \in cp(x)$ **do** $cn(v) \leftarrow cn(v) \cup \{x\}$;
10 $\quad x' \leftarrow$ the immediate predecessor of x according to $\prec$;
11 $\quad$ **if** $x' \notin Q$ **then** Push x' into Q;

predecessors $cp(u)$ to be the neighbors $v \in nbr(u)$ satisfying $v \prec u$ (Line 3). We also initialize a FIFO queue Q (Line 4) to process vertices in a bottom-up manner. All vertices without lower-ranked successors (i.e., $|\mathbb{S}(v)| = 1$) are added to Q (Line 5) as starting points. In each iteration, we pop a vertex x from Q (Line 7) and propagate convex predecessors according to Corollary 5.1 (Line 8). Since all vertices ranked lower than x have already been processed, we add x as a convex neighbor to each of its convex predecessors (Line 9). Finally, x's immediate predecessor under $\prec$ is enqueued for further processing (Lines 10-11).

Lemma 5.2. *Algorithm 3 correctly computes the convex neighbors for all vertex of V in $O\big(\sum_{v \in V} |cp(v)|^2\big)$ total time.*

5.3 Handling Object Deletion

When an object o is removed from $\mathcal{M}$, we may need to update $\{\mathcal{M}_k^{\succeq}(c)\}_{c \in V}$ stored in our index. For a vertex $c \in V$, if $o \notin \mathcal{M}_k^{\succeq}(c)$, it is obvious that $\mathcal{M}_k^{\succeq}(c)$ does not need to be updated. If $o \in \mathcal{M}_k^{\succeq}(c)$, then we need to try to replace o in $\mathcal{M}_k^{\succeq}(c)$ with another object of $\mathcal{M}^{\succeq}(c) \setminus \mathcal{M}_k^{\succeq}(c)$. However, scanning all objects of $\mathcal{M}^{\succeq}(c) \setminus \mathcal{M}_k^{\succeq}(c)$ may result in a substantial overhead for index updating. By leveraging Theorem 5.1, we only need to consider objects in $\{\mathcal{M}_k^{\succeq}(v)\}_{v \in cn(c)}$. Moreover, by the same observation as utilized in designing our query processing Algorithm in Section 4.2, for each $v \in cn(c)$, we only need to consider the first object of $\mathcal{M}_k^{\succeq}(v)$ (as ranked by their distances) that is not in $\mathcal{M}_k^{\succeq}(c)$.

As updating $\mathcal{M}_k^{\succeq}(c)$ requires $\{\mathcal{M}_k^{\succeq}(v)\}_{v \in cn(c)}$ to be up-to-date, the deletion update needs to be applied in a bottom-up manner regarding $\prec$. Furthermore, rather than processing all potential vertices $c \in \mathbb{P}(o)$ whose local k nearest neighbors may need to be updated, we restrict our attention to only those satisfying $o \in \mathcal{M}_k^{\succeq}(c)$. To support this, we maintain a doubly linked list that connects all occurrences of the same object o across different $\mathcal{M}_k^{\succeq}(\cdot)$ sets. This allows us to efficiently process only the affected vertices in a bottom-up manner, significantly reducing unnecessary computations. Our index updating algorithm for object deletion is presented in Algorithm 4, which is self-explanatory following the above discussions.

Lemma 5.3. *Given the* CTLD*-index and a deleted object o, the time complexity of Algorithm 4 is $O(|\mathbb{P}(o)| \cdot k \cdot \ell_{max})$, where $\ell_{\max}$ denotes the maximum number of convex neighbors among the vertices of G.*

Algorithm 4: Object-Deletion

Input: Our CTLD-index and a deleted object o
Output: The updated CTLD-index

foreach *vertex* $c \in \mathbb{P}(o)$ *s.t.* $o \in \mathcal{M}_k^{\succeq}(c)$ *in bottom-up order* **do**
  Remove o from $\mathcal{M}_k^{\succeq}(c)$;
  $C \leftarrow \emptyset$;
  foreach $v \in cn(c)$ **do**
    $o' \leftarrow$ the first object in $\mathcal{M}_k^{\succeq}(v)$ that is not in $\mathcal{M}_k^{\succeq}(c)$;
    if o' *exists* **then** $C \leftarrow C \cup \{o'\}$;
  if $C \neq \emptyset$ **then**
    $o' \leftarrow \arg\min_{x \in C} lsd(c, x)$;
    Append o' to the end of $\mathcal{M}_k^{\succeq}(c)$;

5.4 Handling Object Insertion

Updating our CTLD-index for object insertion is much easier than for object deletion. The main observations are as follows. When an object o is inserted, only those vertices $c \in \mathbb{P}(o)$ may update their $\mathcal{M}_k^{\succeq}(\cdot)$ sets. Moreover, $\mathcal{M}_k^{\succeq}(c)$ is updated only if

(1) $|\mathcal{M}_k^{\succeq}(c)| < k$; or
(2) $|\mathcal{M}_k^{\succeq}(c)| = k$ and $lsd(c, o) < lsd(c, o_k)$, where o_k is the object with the largest local shortest distance in $\mathcal{M}_k^{\succeq}(c)$.

Following these ideas, our index updating algorithm for object insertion is described in Algorithm 5, which is self-explanatory.

Algorithm 5: Object-Insertion

Input: Our CTLD-index and an inserted object o
Output: The updated CTLD-index

foreach *vertex* $c \in \mathbb{P}(o)$ *in bottom-up order* **do**
  if $|\mathcal{M}_k^{\succeq}(c)| < k$ **then** Insert o into $\mathcal{M}_k^{\succeq}(c)$;
  else
    $o_k \leftarrow$ the k-th object in $\mathcal{M}_k^{\succeq}(c)$;
    if $lsd(c, o) < lsd(c, o_k)$ **then**
      Remove o_k from $\mathcal{M}_k^{\succeq}(c)$ and insert o into $\mathcal{M}_k^{\succeq}(c)$;

LEMMA 5.4. *Given the* CTLD*-index and an inserted object* o*, the time complexity of Algorithm 5 is* $O(|\mathbb{P}(o)| \cdot k)$.

We may also leverage Theorem 5.1 to stop the updating early. That is, for a vertex c, if none of the vertices of $cn(c)$ updated their $\mathcal{M}_k^{\succeq}(\cdot)$ set, then $\mathcal{M}_k^{\succeq}(c)$ remains intact. Thus, we can use a queue to track the vertices whose $\mathcal{M}_k^{\succeq}(\cdot)$ may change, and only process vertices of the queue; the queue is initialized with o and updated by using convex predecessors. However, our empirical studies show that Algorithm 5 performs better than the early stop version, probably due to better cache efficiency.

6 Faster Index Construction

With the convex neighbors computed and the property proved in Theorem 5.1, we propose a faster index construction algorithm than Algorithm 1 in this section, by utilizing the convex neighbors.

We observe that convex neighbors are independent to the candidate object set $\mathcal{M}$ and the integer k. That is, convex neighbors remain the same if we change $\mathcal{M}$ or k. As a result, we could set $\mathcal{M}$ to be V and k to be n in Theorem 5.1 to get the following corollary.

Corollary 6.1. *For any vertex $c \in V$, the local shortest distances $\{lsd(c,u) \mid u \in \mathbb{S}(c)\}$ can be computed from*

$$\bigcup_{v \in cn(c)} \big\{\big(u, lsd(c,v) + lsd(v,u)\big) \mid u \in \mathbb{S}(v)\big\} \tag{7}$$

by removing duplicate vertices and preferring shorter distances.

Thus, we can compute $\{lsd(c,u) \mid u \in \mathbb{S}(c)\}$ for vertices $c \in V$ in a bottom up manner regarding $\prec$. Then, when computing $\{lsd(c,u) \mid u \in \mathbb{S}(c)\}$ by Equation (7), the distance $lsd(v,u)$ is already known. However, $lsd(c,v)$ used in Equation (7) cannot be computed in this way. Fortunately, we can compute such $lsd(c,v)$ during the process of obtaining convex neighbors by following the proof of Lemma 5.1.

Algorithm 6: Index-Cons$^+$

Input: A graph $G = (V,E)$, an integer k and candidate objects $\mathcal{M}$
Output: A cut tree T_G of G, convex neighbors, and associated local shortest distances and local k nearest neighbors

1 Run Algorithm 1 (without Lines 2–4) to build the cut tree T_G;
2 **foreach** $v \in V$ *and* $u \in \mathbb{S}(v)$ **do** $lsd(v,u) \leftarrow \infty$;
3 **foreach** $v \in V$ **do** $lsd(v,v) \leftarrow 0$;
4 Run Lines 1–3 of Algorithm 3;
5 **foreach** $u \in V$ *and* $v \in cp(u)$ **do** $lsd(v,u) \leftarrow \phi(v,u)$;
6 Run Lines 4–11 of Algorithm 3, and at Line 8 of Algorithm 3 also execute $lsd(v,u) \leftarrow \min\{lsd(v,u), lsd(v,x) + lsd(u,x)\}$;
7 **foreach** $c \in V$ *in bottom up manner regarding* $\prec$ **do**
8 **foreach** $v \in cn(c)$ *and* $u \in \mathbb{S}(v)$ **do**
9 $lsd(c,u) \leftarrow \min\{lsd(c,u), lsd(c,v) + lsd(v,u)\}$
10 Compute $\mathcal{M}_k^{\succeq}(c)$ by using a similar algorithm as Algorithm 2 but leveraging Theorem 5.1;

The pseudocode of our faster index construction algorithm is shown in Algorithm 6. Line 1 constructs the cut tree T_G by invoking Algorithm 1 but without running Dijkstra's algorithm. Lines 2–3 initialize the local shortest distances. Lines 4–6 compute the convex neighbors for each vertex $v \in V$, and at the same time $\{lsd(c,v)\}_{c \in V, v \in cn(c)}$. Note that, such $lsd(c,v)$ computed after Line 6 may be not the local shortest distance between c and v in $G_{\succeq c}$; it is actually the length of the *local shortest convex path* between c and v in $G_{\succeq c}$, and a local shortest path may be not convex. Nevertheless, by examining the proof of Theorem 5.1, we can see that it is sufficient to use such distances in Equation (7). Moreover, we remark that if none of the local shortest paths betweens c and v is convex, then the correct local shortest distance between c and v will be computed at Line 9 by following the proof of Theorem 5.1. Lines 8–9 compute local shortest distances $\{lsd(c,u) \mid u \in \mathbb{S}(c)\}$ by following the above discussions, while Line 10 computes $\mathcal{M}_k^{\succeq}(c)$ by using a similar algorithm as Algorithm 2 but leveraging Theorem 5.1.

Table 3. Dataset statistics

Dataset	Region	n	m	Type
NY	New York City	264,346	733,846	Undirected
BAY	San Francisco	321,270	800,172	Undirected
COL	Colorado	435,666	1,057,066	Undirected
FLA	Florida	1,070,376	2,712,798	Undirected
CAL	California	1,890,815	4,657,742	Undirected
LKS	Great Lakes	2,758,119	6,885,658	Undirected
E-US	Eastern USA	3,598,623	8,778,114	Undirected
W-US	Western USA	6,262,104	15,248,146	Undirected
CTR	Central USA	14,081,816	34,292,496	Undirected
USA	United States	23,947,347	58,333,344	Undirected
SJC	San Joaquin County	18,263	23,874	Directed
SF	San Francisco	174,956	223,001	Directed
NA	North America	175,813	179,179	Directed

Theorem 6.1. *The time complexity of Algorithm 6 is* $O\Big(m \log n \cdot (\log n + w) + \sum_{v \in V} \big(|cp(v)| \cdot (|cp(v)| + |\mathbb{S}(v)|) + |cn(v)| \cdot k \cdot \log |cn(v)|\big)\Big)$.

7 Experiments

We conduct extensive experiments to evaluate our proposed methods against the state-of-the-art methods in this section.

Datasets. We evaluate the approaches on 13 real-world road networks, including 10 undirected and 3 directed graphs. Table 3 summarizes the details about these datasets. The first 10 datasets are available from DIMACS[1], and others are sourced from SpatialDataset[2]. Most of our experiments focus on the 10 undirected graphs, while the 3 directed graphs are used in **Exp-13** and **14**. In each road network, vertices represent intersections between roads, and edges correspond to roads or road segments. Each edge has an associated weight, representing the distance between endpoints. DIMACS also provides the vertex coordinates required by GLAD.

Environment. All algorithms are implemented in C++ and compiled with g++13.3.0 using the `-O3` flag. Experiments on the CTR and USA datasets are conducted on a Linux machine with dual Intel(R) Xeon(R) Gold 6342 2.8GHz CPUs and 200GB of RAM, while the remaining experiments are performed on a Linux machine with an Intel(R) Xeon(R) W5-2445 CPU and 64GB of RAM.

Baseline Methods. We compare our algorithms with three state-of-the-art solutions GLAD [24], TEN (TEN-index) [36] and KNN (KNN-index) [41] that were discussed in Section 3. Their code was kindly provided by their authors. For GLAD, we use the default grid size of $100m \times 100m$, as suggested in [24].

Parameter Settings. Following prior work [24, 36, 41], we randomly generate candidate objects in each dataset with a specified density $\theta = |\mathcal{M}|/|V|$. The candidate density θ and the query parameter k used in our experiments are shown in Table 4, where the default values are highlighted in bold. The choices of k is aligned with prior work on kNN queries over road networks [24, 33, 36].

7.1 Experimental Results

Exp-1: Query Processing Time by Varying k. In this experiment, we evaluate our CTLD-Query

[1]http://www.dis.uniroma1.it/challenge9/download.shtml
[2]https://users.cs.utah.edu/~lifeifei/SpatialDataset.htm

Table 4. Parameter settings

Parameters	Description	Values
k	number of nearest neighbors	10, **20**, 30, 40
θ	object density i.e., $\lvert\mathcal{M}\rvert/\lvert V\rvert$	0.1, 0.01, **0.001**, 0.0001

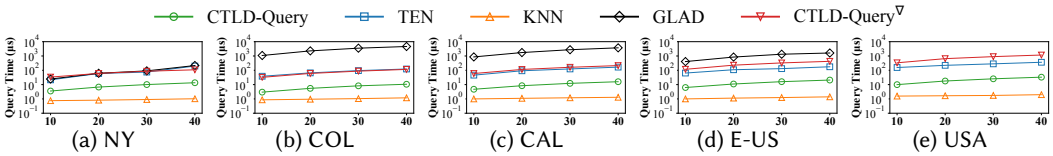


Fig. 3. Query processing time by varying k ($\theta = 10^{-3}$).

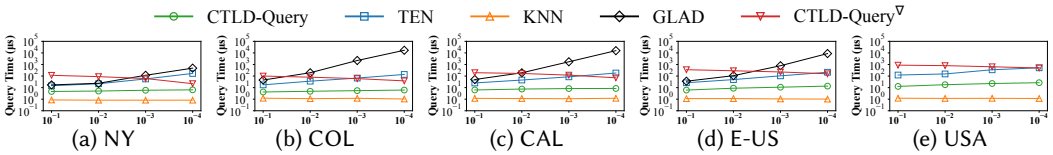
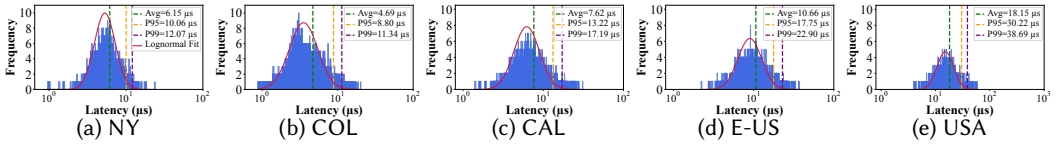


Fig. 4. Query processing time by varying θ ($k = 20$).

(presented in Algorithm 2) against state-of-the-art solutions TEN, KNN, and GLAD, by varying k. We also include a comparison with a naive query processing baseline (denoted as CTLD-Query$^{\nabla}$) for our index, which was described at the beginning of Section 4.2. For each dataset, we randomly generate 10,000 query vertices and report the average processing time of each algorithm. As shown in Fig. 3, CTLD-Query is only slower than KNN that full materializes the kNN results for every vertex, enabling optimal query latency. However, KNN suffers from high update costs (as shown in our subsequent experiments), performing up to 100× slower than our approach. Across all datasets, CTLD-Query consistently outperforms TEN by up to 10×, and achieves up to 10× speedup over GLAD on small networks and 100× on larger datasets. On small networks such as NY and BAY, GLAD and TEN perform comparably. As the graph size increases, TEN scales better and eventually outperforms GLAD by up to 10×. CTLD-Query$^{\nabla}$ achieves lower query latency than TEN in some cases (e.g., BAY), and outperforms GLAD in most cases. This highlights the benefit of our cut tree-based index and clearly shows that storing local distances has no negative impact on query processing time. GLAD is omitted from this and also following experiments for the USA dataset due to its index construction exceeds memory limit.

Exp-2: Query Processing Time by Varying θ. In this experiment, we compare the algorithms' query processing time by varying the density $\theta = |\mathcal{M}|/|V|$ with $\theta \in \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$. For each dataset, we generate four groups of objects based on the specified densities and also randomly generate 10,000 queries. The average query processing time of each method is reported in Fig. 4. The generally trend is similar to that of Fig. 3. In addition, the query processing time of TEN and GLAD increases significantly as the candidate density θ decreases, while our method remains much faster and more stable. As density decreases, our naive query approach becomes faster due to fewer candidate objects to examine, and even outperforms TEN. This contrasts with the other algorithms, since the sparser candidate distribution forces them to explore larger portions of the search space to identify the nearest neighbors.

Exp-3: Tail Latency of Query Time. To analyze the efficiency of our query algorithm in more detail, we record the latency of the 10,000 queries and visualize their distribution in Fig. 5. We fit the measured latencies to a lognormal distribution, shown as the red curve. The results show that the latency distribution can be well approximated by a lognormal shape, suggesting that query latencies of Algorithm 2 are concentrated around the mean with low tail latency. We also report



(a) NY (b) COL (c) CAL (d) E-US (e) USA

Fig. 5. Distribution of query latencies.

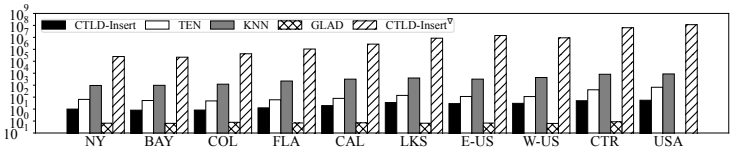


Fig. 6. Insertion time (μs).

Table 5. Average deletion time (ms) on the two largest datasets

Dataset	TEN	KNN	CTLD-Delete	CTLD-Delete$^{\nabla}$
CTR	56.08	30.73	0.34	64106.51
USA	65.48	36.62	0.47	115349.20

the percentile values and the average query time (green dashed line), where P95 (yellow dashed line) and P99 (purple dashed line) represent the latency thresholds below which 95% and 99% of all query times fall, respectively. The P99 latency is about twice the average, as expected, because a few worst-case queries require exploring larger search spaces. Overall, the query latencies are stable, exhibiting only limited tail effects.

Exp-4: Object Insertion. In this experiment, we evaluate the performance of our insertion algorithm (CTLD-Insert, Algorithm 5) against KNN, TEN and GLAD. We also include a naive baseline, denoted as CTLD-Insert$^{\nabla}$, which is discussed in Section 5.1. To generate insertions, we randomly select a vertex u and insert it into the object set if $u \notin \mathcal{M}$. This process is repeated until 10,000 insertions are generated for each dataset. The average insertion time is reported in Fig. 6. GLAD achieves the fastest insertion time by simply adding new objects to the candidate list of the corresponding grid, though this simplicity comes at the cost of query efficiency. CTLD-Insert$^{\nabla}$ incurs the highest insertion overhead, as it recomputes $\mathcal{M}_k^{\succeq}(\cdot)$ for each vertex in $\mathbb{P}(o)$ given an inserted object o. KNN is more than 100× slower than CTLD-Insert, as it needs to update the materialized kNN results for many vertices to maintain query correctness after each insertion. Our method, CTLD-Insert, achieves up to 10× faster insertion performance compared to TEN.

Exp-5: Object Deletion. In this experiment, we evaluate the performance of Algorithm 4 (denoted as CTLD-Delete). We also include the naive deletion approach, discussed in Section 5.1, represented as CTLD-Delete$^{\nabla}$. We generate 10,000 random deletions for objects of $\mathcal{M}$ for CTR and USA and report the average deletion time. We only include the two largest graphs, because other smaller graphs do not have enough objects for 10,000 deletions under the $\theta = 10^{-3}$ setting. As shown in Table 5, our deletion algorithm achieves the best performance, outperforming KNN by more than 70× and TEN by more than 130×. For TEN on the USA dataset, deletion takes 65.48 ms compared to just 0.67 ms for insertion; a similar trend is observed on CTR. This imbalance indicates that when deletions dominate, the update time of TEN increases significantly, whereas fewer deletions lead to a lower overall update cost.

Exp-6: Object Update. We evaluate the performance of the update algorithms with mixed-updates (i.e., including both insertion and deletion) in this experiment. CTLD-Update combines CTLD-Delete and CTLD-Insert, while CTLD-Update$^{\nabla}$ relies on CTLD-Delete$^{\nabla}$ and CTLD-Insert$^{\nabla}$. To generate updates, we randomly select a vertex u and apply an insertion or deletion, skipping deletions

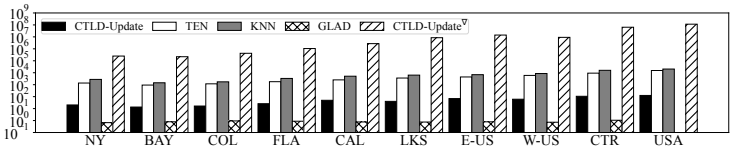


Fig. 7. Update time (μs).

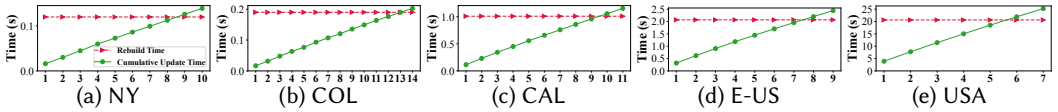


Fig. 8. Batch update performance against rebuilding $\mathcal{M}_k^{\succeq}(\cdot)$ (vary batch size $x \cdot |\mathcal{M}|$).

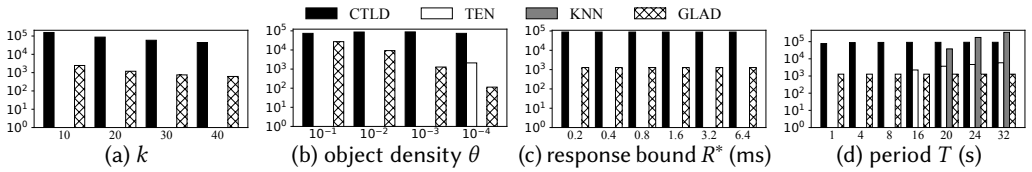


Fig. 9. Throughput (LKS, BUA+QF).

if $u \notin M$ and insertions if $u \in M$. For each dataset, we repeat this process to generate 10,000 updates. The average time per update operation is reported in Fig. 7. CTLD-Update is only slower than GLAD. In contrast, CTLD-Update$^{\nabla}$ exhibits the highest cost, as it recomputes $\mathcal{M}_k^{\succeq}(\cdot)$ given a large update space. Update performance also depends on the ratio of insertions and deletions. In the insertion-only experiment, TEN achieves 0.41 ms on CTR and 0.67 ms on USA, compared to 8.19 ms and 8.69 ms for KNN. However, TEN incurs higher deletion costs than KNN. In the mixed-update experiment, where insertions and deletions are randomly generated, TEN still outperforms KNN, achieving roughly 1.5× higher update performance. CTLD-Insert achieves insertion costs of 0.049 ms on CTR and 0.055 ms on USA, approximately 10× faster than TEN and 100× faster than KNN. Similarly, our deletion performance is approximately 100× faster than TEN and KNN. These results demonstrate that our update performance consistently outperforms TEN by about 10× and KNN by about 100×, regardless of the insertion-deletion ratio.

Exp-7: Batch Updating vs. Reconstruction. Although our primary focus is on the *online updating* setting, we also evaluate the efficiency of our algorithms in the *batch updating* setting and compare their performance with that of reconstructing the index from scratch. For batch updating, we consecutively invoke our insertion algorithm for each object insertion and our deletion algorithm for each object deletion. We generate batches of varying sizes, with each batch containing an equal number of insertions and deletions to maintain a constant number of object. The results are shown in Fig. 8, where the *rebuild time* refers to the time required to recompute $\mathcal{M}_k^{\succeq}(\cdot)$ at the end of each batch. Note that all other components of our index remain unchanged during updates. The x-axis represents the batch size, where each point x corresponds to a batch containing $x \cdot |\mathcal{M}| = x \cdot 10^{-3}n$ updates. As shown in Fig. 8, the threshold at which the batch updating time surpasses the rebuild time remains approximately 8×–14× larger than the candidate set size on small graphs and still 4×–8× larger on large graphs. These results demonstrate that our update algorithms scale gracefully and maintain high efficiency even under extensive updates.

Exp-8: Adaptability. Following [33, 36, 41], we evaluate the adaptability of the algorithms' query throughput under a wide range of update workloads, from low to high. Figures 9 and 10 report the throughput as defined by Equation (1) and Equation (2), respectively, under the BUA+QF and RUA+FCFS scenarios on the LKS dataset. Algorithms with zero throughput are omitted from the

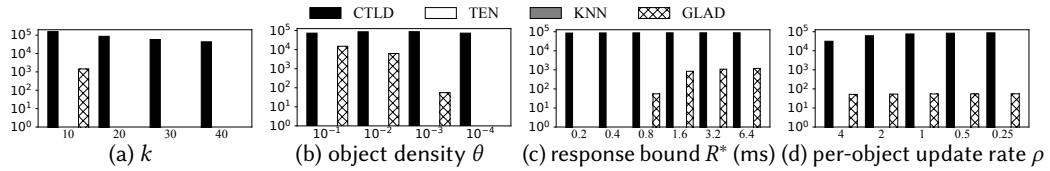


Fig. 10. Throughput (LKS, RUA+FCFS).

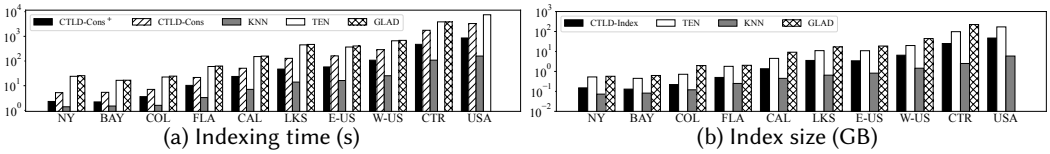


Fig. 11. Indexing time and index size.

figures. Meaning of the parameters R^*, T and ρ and their default values can be found in Table 2. The results show that CTLD consistently achieves high throughput across all scenarios, demonstrating superior adaptability and a balanced trade-off between query and update costs. Notably, CTLD sustains high throughput even under heavy workloads, e.g., when $\rho = 4$ (corresponding to 11,032 updates per second). GLAD remains applicable in many settings but fails under RUA-FCFS when R^* decreases, k increases, or θ drops, as its query time t_q exceeds R^*. For example, the query time of GLAD at $k = 20$ increases to around 0.9 ms, exceeding $R^* = 0.8$ ms. Nevertheless, GLAD is still suited for scenarios with less stringent query latency requirements. We also observe that KNN and TEN achieve non-zero throughput only under extremely low update workloads (e.g., BUA-QF with $T = 24$ and update rate ≈ 115 updates/s). KNN stores exact results for each vertex, which allows it to outperform CTLD under BUA-QF when $T \geq 24$. However, their throughput degrades rapidly as update load grows. For example, with $\theta = 10^{-3}$, $k = 20$, and about 2,758 objects on LKS, KNN requires 6.93 ms per update. At an update rate of $\rho = 0.25$ per object per second (689.5 updates/s), updates would take 478% of the available time budget per second, resulting in zero throughput.

In summary, CTLD achieves the most favorable balance between query efficiency and update scalability, making it highly adaptable to dynamic workloads. In contrast, TEN and KNN are suitable only for light update workloads, while GLAD is more suited to scenarios with relaxed query response requirements.

Exp-9: Indexing Time and Index Size. We report indexing time and index size in Fig. 11; GLAD runs out of memory on USA. CTLD-Cons corresponds to the execution time of Algorithm 1 and Algorithm 3, while CTLD-Cons$^+$ corresponds to that of Algorithm 6. GLAD and TEN exhibit similar indexing time and index size, as both rely on the H2H-index, but the additional per-grid information in GLAD makes its index slightly larger than that of TEN. KNN achieves the fastest indexing time, since it does not compute the large number of local distances as required by our index. On small datasets, CTLD-Cons is 1.5× slower than CTLD-Cons$^+$, and up to 3.5× slower on large datasets, due to its use of Dijkstra's search from each vertex v within $G_{\succeq v}$ to compute distances.

Exp-10: Different Components of Our Index Construction Algorithm. Table 6 breaks down the running time of CTLD-Cons$^+$ by components, with proportion of total time shown in parentheses. This confirms that the relative proportions of the three components remain stable even as the dataset size increases. Across all datasets, building T_G (Line 1 of Algorithm 6) consistently dominates the indexing time, accounting for approximately $70 - 76\%$ of the total. The computation of convex neighbors and local distances (Lines 2–9 of Algorithm 6) contributes around $20 - 24\%$, while that of $\mathcal{M}_k^{\succeq}(\cdot)$ (Lines 7 and 10) incurs the smallest cost, generally below 5%.

Exp-11: Cut Trees With and Without Shortcuts. As remarked at the end of Section 4.1, our cut

Table 6. Running time (s) of index construction components, with percentage of total time in parentheses.

	Build T_G (s)	Calc. lsd, *cn*	Calc. $\mathcal{M}_k^{\succeq}$	Total
NY	1.81 (70.1%)	0.65 (25.2%)	0.12 (4.7%)	2.58
COL	2.82 (75.0%)	0.75 (19.9%)	0.19 (5.1%)	3.76
CAL	18.06 (74.6%)	5.14 (21.2%)	1.01 (4.2%)	24.21
E-US	44.56 (75.7%)	12.21 (20.8%)	2.07 (3.5%)	58.84
USA	634.16 (73.9%)	203.69 (23.7%)	20.64 (2.4%)	858.49

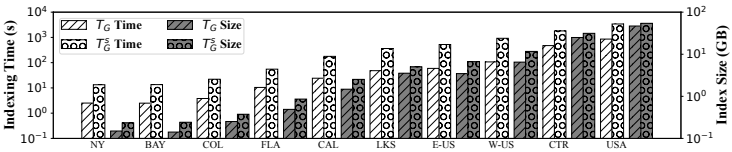


Fig. 12. Comparison of T_G and its variant T_G^s with shortcuts

Table 7. Indexing time (s), index size (GB) and average update time (ms) when varying k

	CTR			USA		
k	Build	Size	Update	Build	Size	Update
20	474.15	25.167	0.07	858.36	47.826	0.12
200	477.64	25.172	0.70	860.72	47.834	1.22
2,000	483.96	25.181	28.36	867.86	47.850	54.23
20,000	490.94	25.188	1518.82	887.60	47.867	1215.36

(a) Indexing time and size (b) Query and update

Fig. 13. Performance of CTLD on directed road networks.

tree differs from that proposed by [21], which introduces shortcuts — within the two subgraphs formed after removing a vertex cut — to preserve global shortest distances within those subgraphs. In this experiment, we compare our index with a variant that incorporates such shortcuts following the method of [21], denoted as T_G^s. Fig. 12 presents the indexing time and index size of both approaches across all datasets. Here, note that T_G and T_G^s refer to the complete indices built from their respective cut trees, rather than the cut trees alone. We observe that the index size of T_G is consistently smaller than that of T_G^s across all datasets, indicating that the inclusion of shortcuts substantially increases the successor sets $\mathbb{S}(v)$ for vertices $v \in V$. Furthermore, the results show that constructing T_G can be up to an order of magnitude faster than constructing T_G^s. This demonstrates the efficiency of our shortcut-free construction and further validates our design choice.

Exp-12: Effect of k on Index Performance. As discussed in Section 4.1, our index is built for a predefined k and supports queries for any $k' \leq k$. In this experiment, we evaluate the effect of k on index performance. The results on CTR and USA with $k \in \{20, 200, 2000, 20000\}$ are reported in Table 7. We observe that both the indexing time and index size increase only slightly as k grows; this is because they are dominated by the cut tree construction and the storage of *lsd*. In contrast, the average update time rises sharply with larger k, because a modified object is likely to appear in more local kNNs $\mathcal{M}_k^{\succeq}(\cdot)$ as k increases. Hence, the index should be built with a small k sufficient to support all user queries.

	SJC	SF	NA
On Vertices	0.52	1.29	1.11
On Edges	0.54	1.31	1.15

Table 8. Query time (μs) for objects on vertices vs. edges.

Exp-13: Performance of CTLD on Directed Graphs. In this experiment, we evaluate our algorithms on the three directed graphs in Table 3. We first construct the cut tree by treating the graph G as undirected, and then incorporate directional information into the data structures. For example, lsd is replaced by lsd_f (for forward direction) and lsd_b (for backward direction). For any two vertices v and u, the shortest distance from v to u in G can be obtained as $dist(v,u) = \min_{c \in \mathbb{P}(v) \cap \mathbb{P}(u)} lsd_b(c,v) + lsd_f(c,u)$. Please refer to the online Supplementary Material for details of the extension. To the best of our knowledge, none of the existing methods' implementations support directed graphs; thus, they are excluded from this experiment. Fig. 13 presents the indexing time, index size, and average query and update time for the directed version of CTLD (denoted with suffix "(D)") and the undirected version ("(U)"), which ignores edge directions and retains the edge with the smaller weight. As shown in Fig. 13, the directed version achieves comparable or lower indexing time and faster query/update performance than the undirected version. The performance difference primarily arises from the sparsity of the datasets: when represented as directed graphs, the overall connectivity decreases, thereby reducing the search space during indexing and querying. As expected, the index size of the directed version is slightly larger due to the need to store both forward and backward local distances.

Exp-14: Objects Located on Edges. In this experiment, we still use the three directed graphs, but with objects located on edges. Specifically, objects are randomly generated under the default density, each assigned to a random edge $\langle u,v \rangle \in E$ with a random offset in $[0, \phi(\langle u,v \rangle))$ measured from the source u. The only modification required during index construction concerns $\mathcal{M}_k^{\succeq}(\cdot)$, which remains defined on the vertices. In computing $\mathcal{M}_k^{\succeq}(u)$, we additionally consider the objects located on the outgoing edges of u. For a query q located on edge $\langle u,v \rangle$, we simply issue a query using vertex v and then combine the results with the objects on edge $\langle u,v \rangle$ to obtain the kNN for q. Table 8 reports the average query processing time for the scenarios of objects "on vertices" and "on edges"; the existing algorithms are omitted as their implementations do not support this setting. We observe that the difference is negligible.

8 Conclusion

In this paper, we studied the kNN search problem on road networks under dynamic object updates. Existing solutions fail to achieve a good tradeoff between query and update efficiency, resulting in limited throughput. To address these limitations, we proposed a local subgraph-based kNN framework that eliminates the reliance on global structures, enabling efficient index maintenance under dynamic updates. We designed an efficient query processing algorithm that incrementally traverses the index to produce the kNN results. We formally proved the correctness of our local subgraph-restricted kNN indexing approach. Our extensive experiments demonstrated that the proposed method achieves superior throughput and strong adaptability across various dynamic workloads.

Acknowledgments

Lijun Chang is supported by ARC DP220103731. Dong Wen is supported by ARC DP230101445 and ARC DE240100668. Dian Ouyang is supported by NSFC No.62502105.

References

[1] [n.d.]. Airbnb. https://www.airbnb.com/.
[2] [n.d.]. Booking. https://www.booking.com/.
[3] [n.d.]. Dianping. https://www.dianping.com/.
[4] [n.d.]. DiDi. https://www.didiglobal.com..
[5] [n.d.]. DoorDash. https://www.doordash.com..
[6] [n.d.]. How Do I Handle Real-Time Tracking In My Delivery App? https://thisisglance.com/learning-centre/how-do-i-handle-real-time-tracking-in-my-delivery-app.
[7] [n.d.]. Lyft. https://www.lyft.com/.
[8] [n.d.]. Meituan. https://www.meituan.com..
[9] [n.d.]. Meituan reports 3.36M average monthly active riders. https://www.techinasia.com/news/meituan-reports-336m-average-monthly-active-riders.
[10] [n.d.]. OpenRice. https://www.openrice.com/.
[11] [n.d.]. Trip. https://www.trip.com/.
[12] [n.d.]. Uber. https://www.uber.com/.
[13] [n.d.]. Uber Eats. https://www.ubereats.com..
[14] [n.d.]. Yelp. https://www.yelp.com/.
[15] Beijing Youth Daily. 2025. Beijing will build 1,000 super charging stations this year. https://www.china-news-online.com/lang/English/4282049.html.
[16] YH Chang, Jia-Shung Wang, and Richard CT Lee. 1994. Generating all maximal independent sets on trees in lexicographic order. *Information sciences* 76, 3-4 (1994), 279–296.
[17] Hyung-Ju Cho and Chin-Wan Chung. 2005. An efficient and scalable approach to CNN queries in a road network. In *International Conference on VLDB*, Vol. 2. International Conference on VLDB, 865–876.
[18] Hyung-Ju Cho, Se Jin Kwon, and Tae-Sun Chung. 2013. A safe exit algorithm for continuous nearest neighbor monitoring in road networks. *Mobile Information Systems* 9, 1 (2013), 37–53.
[19] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. 2009. Efficient continuous nearest neighbor query in spatial networks using euclidean restriction. In *International Symposium on Spatial and Temporal Databases*. Springer, 25–43.
[20] Edsger W Dijkstra. 2022. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: his life, work, and legacy*. 287–290.
[21] Muhammad Farhan, Henning Koehler, Robert Ohms, and Qing Wang. 2023. Hierarchical Cut Labelling-Scaling Up Distance Queries on Road Networks. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–25.
[22] Gasgoo. 2022. Beijing eyes 700,000 EV charging piles by 2025. https://autonews.gasgoo.com/new_energy/70020023.html.
[23] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International workshop on experimental and efficient algorithms*. Springer, 319–333.
[24] Dan He, Sibo Wang, Xiaofang Zhou, and Reynold Cheng. 2019. An efficient framework for correctness-aware knn queries on road networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1298–1309.
[25] International Energy Agency. 2025. Global EV Outlook 2025. https://www.iea.org/reports/global-ev-outlook-2025 Licence: CC BY 4.0.
[26] Wei Jiang, Guanyu Li, Mei Bai, Bo Ning, Xite Wang, and Fangliang Wei. 2023. Graph-Indexed k NN Query Optimization on Road Network. *Electronics* 12, 21 (2023), 4536.
[27] Wei Jiang, Fangliang Wei, Guanyu Li, Mei Bai, Yongqiang Ren, and Jingmin An. 2021. Tree index nearest neighbor search of moving objects along a road network. *Wireless Communications and Mobile Computing* 2021, 1 (2021), 2050489.
[28] Mohammad Kolahdouzan and Cyrus Shahabi. 2004. Voronoi-based k nearest neighbor search for spatial network databases. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 840–851.
[29] Mohammad R Kolahdouzan and Cyrus Shahabi. 2005. Alternative solutions for continuous k nearest neighbor queries in spatial network databases. *GeoInformatica* 9 (2005), 321–341.
[30] Ken C.K. Lee, Wang-Chien Lee, Baihua Zheng, and Yuan Tian. 2012. ROAD: A New Spatial Object Search Framework for Road Networks. *IEEE Transactions on Knowledge and Data Engineering* 24, 3 (2012), 547–560. https://doi.org/10.1109/TKDE.2010.243
[31] Chuanwen Li, Yu Gu, Jianzhong Qi, Jiayuan He, Qingxu Deng, and Ge Yu. 2018. A gpu accelerated update efficient index for knn queries in road networks. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 881–892.

[32] Jiajia Li, Cancan Ni, Dan He, Lei Li, Xiufeng Xia, and Xiaofang Zhou. 2023. Efficient k NN query for moving objects on time-dependent road networks. *The VLDB Journal* 32, 3 (2023), 575–594.
[33] Siqiang Luo, Ben Kao, Guoliang Li, Jiafeng Hu, Reynold Cheng, and Yudian Zheng. 2018. Toain: a throughput optimizing adaptive index for answering dynamic k nn queries on road networks. *Proceedings of the VLDB Endowment* 11, 5 (2018), 594–606.
[34] Kyriakos Mouratidis, Man Lung Yiu, Dimitris Papadias, and Nikos Mamoulis. 2006. Continuous nearest neighbor monitoring in road networks. (2006).
[35] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In *Proceedings of the 2018 International Conference on Management of Data*. 709–724.
[36] Dian Ouyang, Dong Wen, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Progressive top-k nearest neighbors search in large road networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1781–1795.
[37] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. 2003. Query processing in spatial network databases. In *Proceedings 2003 VLDB Conference*. Elsevier, 802–813.
[38] Cyrus Shahabi, Mohammad R Kolahdouzan, and Mehdi Sharifzadeh. 2002. A road network embedding technique for k-nearest neighbor search in moving object databases. In *Proceedings of the 10th ACM international symposium on advances in geographic information systems*. 94–100.
[39] Bilong Shen, Ying Zhao, Guoliang Li, Weimin Zheng, Yue Qin, Bo Yuan, and Yongming Rao. 2017. V-tree: Efficient knn search on moving objects with road-network constraints. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 609–620.
[40] Paul Tassi. 2016. The New Pokémon GO Update Has Killed 'Nearby' Tracking Completely. https://www.forbes.com/sites/insertcoin/2016/07/30/the-new-pokemon-go-update-has-killed-nearby-tracking-completely. Accessed: 2025-06-20.
[41] Yiqi Wang, Long Yuan, Wenjie Zhang, Zi Chen, Xuemin Lin, and Qing Liu. 2024. Simpler is More: Efficient Top-K Nearest Neighbors Search on Large Road Networks. *Proc. VLDB Endow.* 17, 13 (Sept. 2024), 4683–4695. https://doi.org/10.14778/3704965.3704975
[42] Jinbo Xu, Feng Jiao, and Bonnie Berger. 2005. A tree-decomposition approach to protein structure prediction. In *2005 IEEE Computational Systems Bioinformatics Conference (CSB'05)*. IEEE, 247–256.
[43] Bolong Zheng, Kai Zheng, Xiaokui Xiao, Han Su, Hongzhi Yin, Xiaofang Zhou, and Guohui Li. 2016. Keyword-aware continuous knn query on road networks. In *2016 IEEE 32Nd international conference on data engineering (ICDE)*. IEEE, 871–882.
[44] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, Lizhu Zhou, and Zhiguo Gong. 2015. G-Tree: An Efficient and Scalable Index for Spatial Search on Road Networks. *IEEE Transactions on Knowledge and Data Engineering* 27, 8 (2015), 2175–2189. https://doi.org/10.1109/TKDE.2015.2399306

Received July 2025; revised October 2025; accepted November 2025