# Simpler is More: Efficient Top-K Nearest Neighbors Search on Large Road Networks

Yiqi Wang
Nanjing University of Science and Technology
The University of New South Wales
yiqi.wang1@unsw.edu.au

Long Yuan*
Nanjing University of Science and Technology
longyuan@njust.edu.cn

Wenjie Zhang
The University of New South Wales
wenjie.zhang@unsw.edu.au

Zi Chen
Nanjing University of Aeronautics and Astronautics
zichen@nuaa.edu.cn

Xuemin Lin
Shanghai Jiaotong University
xuemin.lin@sjtu.edu.cn

Qing Liu
Data61, CSIRO, Australia
q.liu@data61.csiro.au

## ABSTRACT

Top-k Nearest Neighbors ($k$NN) problem on road network has numerous applications on location-based services. As direct search using the Dijkstra's algorithm results in a large search space, a plethora of complex-index-based approaches have been proposed to speedup the query processing. However, even with the current state-of-the-art approach, long query processing delays persist, along with significant space overhead and prohibitively long indexing time. In this paper, we depart from the complex index designs prevalent in existing literature and propose a simple index named KNN-Index. With KNN-Index, we can answer a $k$NN query optimally and progressively with small and size-bounded index. To improve the index construction performance, we propose a bidirectional construction algorithm which can effectively share the common computation during the construction. Theoretical analysis and experimental results on real road networks demonstrate the superiority of KNN-Index over the state-of-the-art approach in query processing performance, index size, and index construction efficiency.

**PVLDB Reference Format:**
Yiqi Wang, Long Yuan, Wenjie Zhang, Xuemin Lin, Zi Chen, Qing Liu. Simpler is More: Efficient Top-K Nearest Neighbors Search on Large Road Networks. PVLDB, 17(13): 4683 - 4695, 2024.
doi:10.14778/3704965.3704975

**PVLDB Artifact Availability:** The source code, data, and/or other artifacts have been made available at https:https://github.com/xiaoqi-7/knn.

## 1 INTRODUCTION

Graphs have been widely used to represent the relationships of entities in many areas [10, 18, 27, 36, 42, 46–50, 52, 53]. Top $k$ nearest neighbors ($k$NN) search on road network is a fundamental operation in location-based services [1, 3, 29]. Formally, given a road network $G(V, E)$, a set of candidate objects $\mathcal{M}$, and a query vertex $u$, $k$NN search identifies $k$ objects in $\mathcal{M}$ with the shortest distance to $u$. $k$NN search finds many important real world applications. For example, in the accommodation booking platforms like *Booking* [4], *Airbnb* [2] and *Trip* [43], an important operation is to show several accommodations closest to the location provided by users. In restaurant-review services, such as *Yelp* [55], *Dianping* [13] and *OpenRice* [30], platforms utilize $k$NN search to present several nearby restaurants to the user. In ride-hailing services like *Uber* [45] and *Didi* [14], several available vehicles near the pickup location are presented before users send the ride-hailing request.


**Figure 1: $k$NN Search in Location-based Service ($k = 3$)**

*Example 1.1.* Figure 1 shows a $k$NN search example in location-based service. Assume that tourists in New York, such as "John" and "Jennie", want to find Starbucks nearby to drink coffee, the location-based service providers like Google Map generally present several candidate stores based on the distance from their locations, which can be modeled as $k$NN search problem. In Figure 1, there are 6 Starbucks stores (marked with $A, B, \cdots, F$), therefore, the candidate object set $\mathcal{M} = \{A, B, C, D, E, F\}$. For "Jennie", the 3NN search returns $\{E, F, B\}$ while the 3NN search for "John" returns $\{C, D, B\}$.

**Motivation.** Given a $k$NN query for vertex $u$, the query can be directly answered by exploring the vertices based on their distance to $u$ using Dijkstra's algorithm [15]. Nevertheless, this method is inefficient, especially when the road network is large and the

---

* Long Yuan is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 13 ISSN 2150-8097.
doi:10.14778/3704965.3704975

candidate objects are far from $u$. Therefore, researchers resort to indexing-based solutions to accelerate query processing [11, 17, 23, 26, 33, 34, 40, 60].

Although existing index-based approaches have made strides in accelerating the query processing, they still suffer from the long query processing delay and their performance is far from optimal. Additionally, these solutions exhibit significant space overhead and prohibitively long indexing times, severely limiting their practical applicability. Take the state-of-the-art approach TEN-Index [33] for $k$NN queries on road networks as an example. The size of TEN-Index on the dataset USA with only 23.95 million vertices and 58.33 million edges (nearly 442.5 MB if an edge is stored by two 4 byte integers) exceeds 160 GB, and it takes more than 5.4 hours to construct the corresponding index. Motivated by these, this paper aims to propose a new index-based solution for $k$NN query that can overcome the shortcomings of existing solutions in query processing performance, index size and index construction.

**A Minimalist $k$NN Index Design.** Revisiting the existing solutions, they generally design a complex index to speedup the query processing. For example, TEN-Index consists of three different parts. Incredibly, as an index for $k$NN query, one of the three parts is even a complete index structure for shortest distance query. The complex-index design leads to the drawbacks of TEN-Index as analyzed in Section 3. This drives us to ask: is this complex-index design thinking really suitable for $k$NN query?

In this paper, we adopt a completely opposite design approach. Going back to the essence of $k$NN search, it only needs to return the $k$ nearest neighbors for the query vertex. Moreover, the $k$ value of the $k$NN search used in real applications is typically not large as users often have limited attention spans and prefer to quickly obtain relevant information to reduce cognitive load and facilitate decision-making [7, 26, 35, 38, 41, 44]. For example, Yelp App [55] provides customers with 20 results every time when searching nearest specific place type, such as restaurant or gas station. A similar strategy is also adopted in other Apps like OpenRice [30] and OpenTable [31]. Therefore, our proposed new index named KNN-Index only simply records the $k$ nearest neighbors of each vertex. The benefits of this minimalist $k$NN index design are twofold: regarding the query processing, the query can be answered *progressively* in *optimal* time. Regarding the space-consumption of the index, only the essential information directly to $k$NN query is stored in the index and the value of $k$ is small in practice, resulting in a *well-bounded* index space.

**New Challenges.** KNN-Index successfully addresses the issues of long query delays and oversized indexes by directly storing the $k$ nearest neighbors for each vertex. However, this strategy leaves the trouble to the index construction as the index structure intuitively implies that we have to explore all the query space before constructing it. A straightforward approach is to compute the $k$ nearest neighbors for each vertex by Dijkstra's algorithm [15]. However, the time complexity of this approach is $O(n \cdot (m + n \log n))$, where $n$ is the number of vertices and $m$ is the number of edges in the road network. Clearly, this approach is impractical to handle large road networks. Another possible approach is to use the existing index like TEN-Index to accelerate the computation of $k$ nearest neighbors for each vertex. Nevertheless, this approach unavoidably induces the drawbacks of existing approaches as discussed above. Overall, the efficiency of the index construction algorithm determines the applicability of our index while it is challenging to design such an efficient index construction algorithm that could outperform existing solutions.

**Our Idea.** The above discussed approaches compute the $k$ nearest neighbors for each vertex independently, which miss the potential opportunities to re-use the intermediate results during the construction. Therefore, we adopt a computation sharing strategy to achieve the efficient index construction. To effectively share the computation, we introduce the concept of *bridge neighbor set* for a vertex $v$ and reveal the hidden relationships between its bridge neighbor set and $k$ nearest neighbors. Following these findings, we design a bridge neighbor preserved graph (BN-Graph) of the input road network with which the bridge neighbor set of a vertex can be easily obtained. Based on BN-Graph, we first propose a bottom-up index construction algorithm in which the intermediate results during the construction can be largely shared and further improve the performance by introducing a bidirectional construction algorithm. Additionally, the given candidate objects $\mathcal{M}$ may be updated in some cases [33], we also design efficient algorithm to incrementally maintain the index for these updates.

**Contributions.** In this paper, we make the following contributions:

*(1) A new attempt at an alternative kNN index design paradigm with a simple yet effective kNN index.* Recognizing the complex index in the existing solutions leads to long query processing delay, oversized index and prohibitive indexing time, we embrace minimalism and design a simple $k$NN index that has a *well-bounded* space and enables *progressive* and *optimal* query processing. To the best of our knowledge, this is the first work that systematically studies such simple yet effective index for $k$NN query.

*(2) Efficient index construction and maintenance algorithms.* Following the designed index, we propose a novel index construction algorithm with which the shortest distance computation regarding a vertex and its top $k$ nearest neighbors can be effectively shared. We also propose index maintenance algorithms to handle object insertion and deletion. We provide time complexity analysis for all proposed algorithms.

*(3) Extensive experiments on real-world road networks.* We extensively evaluate our proposed algorithms on real road networks. Compared with the state-of-the-art approach TEN-Index, experimental results demonstrate that our approach reduces the index space two order of magnitude, speeds up the query time up to two orders of magnitude, and achieves up to two orders of magnitude speedup in index construction.

## 2 PRELIMINARIES

Let $G = (V, E)$ be a connected and weighted graph to represent a real-world road network, where $V(G)$ and $E(G)$ is the set of vertices and edges in $G$, respectively. We use $n = |V(G)|$ (resp. $m = |E(G)|$) to denote the number of vertices (resp. edges) in $G$. For each vertex $v \in V(G)$, the neighbours of $v$, denoted by $\mathsf{nbr}(v, G)$, is defined as $\mathsf{nbr}(v, G) = \{u | (u, v) \in E(G)\}$. The degree of a vertex $v$ is the number of neighbors of $v$, i.e., $\deg(v, G) = |\mathsf{nbr}(v, G)|$. The weight of an edge $(u, v)$ is denoted as $\phi((u, v), G)$. A path $p$ in $G$ is

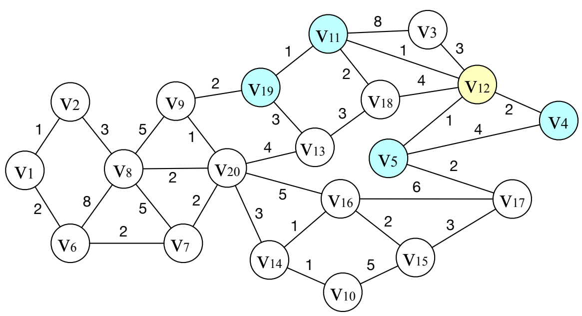



**Figure 2: A Road Network**

a sequence of vertices $p = (v_0, v_1, v_2, \ldots, v_n)$, such that $v_i \in V(G)$ for each $0 \leq i \leq n$. The length of $p$, denoted by $\mathsf{len}(p)$, is the sum for the weight of edges in $p$, i.e., $\mathsf{len}(p) = \sum_{i=1}^{n} \phi(v_{i-1}, v_i)$. Given two vertices $u$ and $v$, the shortest path between $u$ and $v$ in $G$ is a path $p$ from $u$ to $v$ with smallest $\mathsf{len}(p)$. The distance between u and v in G, denoted as $\mathsf{dist}(u, v)$, is the weight of the shortest path between them.

Regarding the given set of candidate objects $\mathcal{M}$, we assume all objects in $\mathcal{M}$ are on vertices following the previous works [17, 26, 33, 40]. In real-world road networks, each object $o \in \mathcal{M}$ may appear on any point of edges. For an object $o$ not on a vertex, we can see $o$ as a vertex object with an offset, and the distance between $o$ and a query vertex $v$ can be computed by mapping $o$ to an adjacent vertex with an offset following the previous works [17, 26, 33, 40] as well. Specifically, assume $o$ is on an edge $(u_o, u'_o)$ with a distance $\phi_o$ to $u'_o$, $q$ is on an edge $(u_q, u'_q)$ with a distance $\phi_q$. The distance between $q$ and $o$ is represented as $\mathsf{dist}(q, o) = \phi_q + \mathsf{dist}(u_q, u_o) + \phi_o$. We denote the $k$NN result of a vertex $u$ as $\mathsf{V_k}(u)$ and define the problem of $k$NN search as follows.

**Problem Definition.** Given a road network $G = (V, E)$, a query vertex $u$, an integer $k$, and a set of candidate objects $\mathcal{M}$ ($|\mathcal{M}| > k$), $\mathcal{M} \subseteq V(G)$), we aim to computes $k$ objects from $\mathcal{M}$, denoted by $\mathsf{V_k}(u)$, such that $\forall v \in \mathsf{V_k}(u)$, $w \in \mathcal{M} \setminus \mathsf{V_k}(u)$, $\mathsf{dist}(u, v) \leq \mathsf{dist}(u, w)$.

*Example 2.1.* Consider the graph $G$ in Figure 2 and assume all vertices are in the candidate object set. For a given query vertex $v_{12}$ and $k = 5$, $V_5(v_{12}) = \{v_{12}, v_5, v_{11}, v_4, v_{19}\}$. The corresponding distances between $v_{12}$ and vertex in $V_5(v_{12})$ are 0, 1, 1, 2 and 2 respectively.

## 3 THE STATE-OF-THE-ART SOLUTION

TEN-Index [33] is the state-of-the-art index-based approach for $k$NN queries on road networks. TEN-Index designs an index based on tree decomposition [37, 54] and H2H-Index [32], which proves superiority over other existing approaches. Specifically,

**Index Structure.** TEN-Index decomposes the input road network into a tree-like structure by tree decomposition [54]. Given the decomposed tree structure, each vertex $u$ has a child vertex set $\mathbb{T}(u)$ and an ancestor vertex set $\mathbb{A}(u)$. Apart from the decomposed tree structure, TEN-Index contains the other two parts: $k$TNN for each vertex $u$ which stores the top $k$ nearest neighbors of $u$ in $\mathbb{T}(u)$ and H2H-Index [32] which is used to compute the shortest distance between $u$ and $v \in \mathbb{A}(u)$.

**Query Processing.** Given a query vertex $u$, for each vertex $v$ in the $k$NN of $u$, there exists a vertex $p$ such that $p \in \mathbb{A}(u) \cup \{u\}$ and $v$ in $k$TNN of $p$. Following this idea, TEN-Index answers the $k$NN query in $k$ rounds. In each $i$ round ($1 \leq i \leq k$), it outputs the top $i$-th result by iterating the vertices in $\mathbb{A}(u) \cup \{u\}$ and computing the corresponding shortest distance through H2H-Index.

**Index Construction.** To construct the index, TEN-Index first decomposes the graph following [54]. With the decomposed tree, $\mathbb{A}(u)$ and $\mathbb{T}(u)$ can be obtained accordingly. After that, TEN-Index builds the H2H-Index based on [32]. At last, the $k$TNN for each vertex is constructed by querying the shortest distance of corresponding vertex pairs through H2H-Index.

**Drawbacks.** Although TEN-Index accelerates the $k$NN query processing on road network, the following drawbacks limit its applicability in practice:

- **Oversized Index.** The size of TEN-Index is generally huge in practice. As verified in our experiments, the size of TEN-Index on USA (only 23, 947, 347 vertices and 58, 333, 344 edges) exceeds 172.80 GB, in which H2H-Index takes 169.23 GB space.
- **Long Query Delay.** To answer a $k$NN query regarding vertex $u$, TEN-Index has to iterate the vertices in $\mathbb{A}(u) \cup \{u\}$ and compute the corresponding shortest distance in $k$ rounds. Moreover, the shortest distance computation is not free, and needs heavy exploration on the H2H-Index. These two factors lead to long query delay of TEN-Index.
- **Prohibitive Indexing Time.** As shown in the above, to construct the index, TEN-Index has to decompose the road network first, and then build the H2H-Index and compute the $k$TNN accordingly. Obviously, the time cost of these procedures are expensive, especially the H2H-Index construction. For the dataset USA, TEN-Index takes 19666s to construct the index, in which H2H-Index consumes 19632s.

## 4 OUR INDEXING APPROACH

According to the above analysis, although the use of H2H-Index accelerates the query processing of TEN-Index, heavily depending on the H2H-Index directly leads to the drawbacks of TEN-Index. This raises a natural question: why do we need an index for shortest distance such as H2H-Index when addressing $k$NN problem? Based on the logic of TEN-Index, partial $k$NN (namely $k$TNN) is maintained for each vertex and H2H-Index is used to refine the partial $k$NN to obtain the final results when processing the query. This motivates us to further ask: Is it necessary to maintain the partial $k$NN? How about maintaining the $k$NN for each vertex directly as an index? In this way, the drawbacks regarding index size and query delay can be totally addressed. Following this idea, we propose the following index and query processing algorithm.

### 4.1 Index Structure and Query Processing

Our index just simply records the $k$NN for each vertex in the graph, which is formally defined as follows:

*Definition 4.1.* (KNN-Index) Given a graph $G$, an integer $k$ and a set of candidate objects $\mathcal{M}$ ($|\mathcal{M}| > k$), for each vertex $v \in G$, KNN-Index records the top-$k$ nearest neighbors of $v$ in $\mathcal{M}$, namely $\mathsf{V_k}(u)$, in the increasing order of their shortest distances from $v$.

*Example 4.2.* Given the graph $G$ in Figure 2, assume the candidate object set is all vertices in $G$ and $k = 5$, the KNN-Index of $G$ is shown in Figure 3. Take $v_8$ as an example, $V_5(v_8) = \{v_8, v_{20}, v_2, v_9, v_1\}$, with shortest distance 0, 3, 3, 4 and 4 respectively.

| $v$ | KNN-Index | $v$ | KNN-Index |
|---|---|---|---|
| $v_1$ | $(v_1,0)(v_2,1)(v_6,2)(v_7,4)(v_8,4)$ | $v_{11}$ | $(v_{11},0)(v_{12},1)(v_{19},1)(v_5,2)(v_{18},2)$ |
| $v_2$ | $(v_2,0)(v_1,1)(v_6,3)(v_8,3)(v_7,5)$ | $v_{12}$ | $(v_{12},0)(v_{11},1)(v_5,1)(v_4,2)(v_{19},2)$ |
| $v_3$ | $(v_3,0)(v_{12},3)(v_5,4)(v_{11},4)(v_4,5)$ | $v_{13}$ | $(v_{13},0)(v_{18},3)(v_{19},3)(v_{11},4)(v_{20},4)$ |
| $v_4$ | $(v_4,0)(v_{12},2)(v_5,3)(v_{11},3)(v_{19},4)$ | $v_{14}$ | $(v_{14},0)(v_{10},1)(v_{16},1)(v_{15},3)(v_{20},3)$ |
| $v_5$ | $(v_5,0)(v_{12},1)(v_{11},2)(v_{17},2)(v_4,3)$ | $v_{15}$ | $(v_{15},0)(v_{16},2)(v_{14},3)(v_{17},3)(v_{10},4)$ |
| $v_6$ | $(v_6,0)(v_1,2)(v_7,2)(v_2,3)(v_{20},4)$ | $v_{16}$ | $(v_{16},0)(v_{14},1)(v_{10},2)(v_{15},2)(v_{20},4)$ |
| $v_7$ | $(v_7,0)(v_6,2)(v_{20},2)(v_9,3)(v_1,4)$ | $v_{17}$ | $(v_{17},0)(v_5,2)(v_{12},3)(v_{15},3)(v_{11},4)$ |
| $v_8$ | $(v_8,0)(v_{20},2)(v_2,3)(v_9,3)(v_1,4)$ | $v_{18}$ | $(v_{18},0)(v_{11},2)(v_{12},3)(v_{13},3)(v_{19},3)$ |
| $v_9$ | $(v_9,0)(v_{20},1)(v_{19},2)(v_7,3)(v_8,3)$ | $v_{19}$ | $(v_{19},0)(v_{11},1)(v_9,2)(v_{12},2)(v_5,3)$ |
| $v_{10}$ | $(v_{10},0)(v_{14},1)(v_{16},2)(v_{15},4)(v_{20},4)$ | $v_{20}$ | $(v_{20},0)(v_9,1)(v_7,2)(v_8,2)(v_{14},3)$ |

**Figure 3:** KNN-Index **of** $G$ **(**$k = 5$**)**

**Query Processing.** Based on our KNN-Index, for a $k$NN query regarding a vertex $v$, we can answer the query directly by retrieving the corresponding items of $v$ in the KNN-Index.

## 4.2 Theoretical Analysis

Following the index structure and query processing algorithm, we have the following theoretical results.

**Optimal Query Processing.** Since our query processing algorithm can answer the query directly by scanning the corresponding items of the query vertex in the KNN-Index, the following theorem exists obviously:

Theorem 4.3. *Given a kNN query, our algorithm takes $O(k)$ time to process the query.*

To answer a $k$NN query, any algorithm needs to output the $k$ results at least, which takes $O(k)$ time. On the other hand, Theorem 4.3 shows the time complexity of our query processing algorithm is $O(k)$. Therefore, the optimality holds.

**Incremental Polynomial Query Processing.** Consider an algorithm that returns several results. Let $k$ be the number of results in the output. An algorithm is said to have incremental polynomial if for all $i \leq k$, the output time of the first $i$ results is bounded by a polynomial function of the input size and $i$ [5]. Since the items for each vertex $v$ in the KNN-Index are recorded in the increasing order of their distance from $v$, we have:

Theorem 4.4. *Given a kNN query regarding v, for every $1 \leq i \leq k$, our algorithm outputs the top i-th nearest neighbor in $O(i)$ time.*

Theorem 4.4 shows that our query processing algorithm is incremental polynomial, indicating that it progressively provides results for a query within a bounded delay. The capability of incremental polynomial query processing is considered as a significant technical contribution of TEN-Index [33] and Theorem 4.4 confirms that our algorithm also possesses this desirable theoretical guarantee.

**Bounded Index Space.** Since KNN-Index only stores the top-$k$ nearest neighbors of each vertex in the road network, we have:

Theorem 4.5. *Given a road network G and an integer k, the size of* KNN-Index *is bounded by $O(n \cdot k)$.*

# 5 INDEX CONSTRUCTION

Based on the structure of KNN-Index, it can be constructed straightforwardly by computing the top $k$ nearest neighbors of each vertex through Dijkstra's algorithm or TEN-Index. However, these approaches are time-consuming and inefficient to handle large road network. In this section, we present our new approach to construct the KNN-Index.

## 5.1 Key Properties of $\mathsf{V_k}(u)$

The above discussed direct approaches using Dijkstra's algorithm or TEN-Index compute the $k$ nearest neighbors for each vertex independently, which misses the potential opportunities to re-use the intermediate results during the index construction. In this section, we introduce two important properties regarding the distance computation, which lays the foundation for our computation-sharing index construction algorithms. We first define:

*Definition 5.1.* **(Bridge Neighbor Set)** Given a vertex $u \in V(G)$, the bridge neighbor set of $u$, denoted by $\mathsf{BNS}(u)$, is the set of $u's$ neighbors $v$ such that the weight of the edge $(u, v)$ is equal to the distance between $u$ and $v$ in $G$, i.e., $\mathsf{BNS}(u) = \{v | v \in \mathsf{nbr}(u, G) \wedge \phi((u, v), G) = \mathsf{dist}((u, v), G)\}$.

*Example 5.2.* Consider $v_8$ in Figure 2. $\mathsf{nbr}(v_8, G) = \{v_2, v_6, v_7, v_{20}, v_9\}$. The shortest path between $v_8$ and $v_9$ is $(v_8, v_{20}, v_9)$, and the distance is $\mathsf{dist}((v_8, v_9), G) = 3$. As $\mathsf{dist}((v_8, v_9), G) \neq \phi((v_8, v_9), G)$, $v_9$ is not in $\mathsf{BNS}(v_8)$. Similarly, $v_6$ and $v_7$ also does not belong to $\mathsf{BNS}(v_8)$. For the graph $G$ in Figure 2, the bridge neighbor set of $v_8$ is $\mathsf{BNS}(v_8) = \{v_2, v_{20}\}$.

Based on Definition 5.1, we have following property regarding the bridge neighbor set of $u$ and its $k$ nearest neighbors:

Property 1. *Given a vertex $u \in V(G)$, $\mathsf{V_k}(u) \subseteq \cup_{v \in \mathsf{BNS}(u)} \mathsf{V_k}(v)$.*

Proof: We prove this property by contradiction. Assume that $w \in \mathsf{V_k}(u)$ but $w \notin \cup_{v \in \mathsf{BNS}(u)} \mathsf{V_k}(v)$. According to Definition 5.1, the shortest path between $w$ and $u$ must pass through one vertex $v \in \mathsf{BNS}(u)$ such that for all $v_i \in \mathsf{V_k}(v)$, $i \in [1, k]$, we have $\mathsf{dist}(v, v_i) < \mathsf{dist}(w, v)$. Therefore, $\mathsf{dist}(u, v) + \mathsf{dist}(v, v_i) < \mathsf{dist}(u, v) + \mathsf{dist}(v, w)$. This implies that there are at least $k$ vertices whose distance to $u$ are smaller than the distance between $u$ and $w$, which contradicts $w \in \mathsf{V_k}(u)$. The proof completes. □

Following Property 1, we have:

Property 2. *Given a vertex $u \in V(G)$, $\mathsf{dist}((u, w), G) = \min_{v \in \mathsf{BNS}(u)} \{\mathsf{dist}((u, v), G) + \mathsf{dist}((v, w), G)\}$ where $w \in \mathsf{V_k}(u)$.*

Proof: According to Definition 5.1, for $\forall w \in \mathsf{V_k}(u)$, each shortest path between $w$ and $u$ must pass through at least one vertex $v \in \mathsf{BNS}(u)$, so we have $\mathsf{dist}(u, w) = \mathsf{dist}(u, v) + \mathsf{dist}(v, w)$. □

Based on Property 1, when a vertex $w \in \mathsf{V_k}(u)$, $w$ must be in $\cup_{v \in \mathsf{BNS}(u)} \mathsf{V_k}(v)$. Moreover, when the bridge neighbor set $\mathsf{BNS}(u)$ of $u$, the distance $\mathsf{dist}(u, v)$ and $\mathsf{V_k}(v)$ ($\mathsf{dist}((v, w), G)$ accordingly where $w \in \mathsf{V_k}(v)$) for all $v \in \mathsf{BNS}(u)$ have been computed, we can compute $\mathsf{dist}((u, w), G)$ for each $w \in \cup_{v \in \mathsf{BNS}(u)} \mathsf{V_k}(v)$ efficiently following Property 2. Obviously, $\mathsf{V_k}(u)$ just selects $k$ vertices from $\cup_{v \in \mathsf{BNS}(u)} \mathsf{V_k}(v)$ with the smallest distance values. Therefore, if we process the vertices in $G$ in a certain order, and when processing each vertex $u$, the vertices $v \in \mathsf{BNS}(u)$ and $\mathsf{V_k}(v)$ have been computed, then $\mathsf{V_k}(u)$ and thereby KNN-Index can be computed efficiently by sharing the computed results. The remaining problem is how to make this idea practically applicable. In next section, we present a bottom-up computation-sharing algorithm, which paves the way to our final index construction algorithm.

## 5.2 A Bottom-Up Computation-Sharing Algorithm

To compute the bridge neighbor set and share the computation effectively, we construct the index based on the bridge neighbor preserved graph $G'$ of the road network $G$, which is defined as:

*Definition 5.3.* **(BN-Graph)** Given a road network $G$, a graph $G'$ is a bridge neighbor preserved graph (BN-Graph) of $G$ if (1) $V(G') = V(G)$; (2) for each edge $(u, v) \in E(G')$, $\phi((u, v), G') = \text{dist}((u, v), G)$; (3) for any two vertices $u, v \in V(G')$, $\text{dist}((u, v), G') = \text{dist}((u, v), G)$.

**Algorithm 1:** SD-Graph-Gen$(G, \pi)$

1 $G' \leftarrow G$;
2 **for each** $w \in V(G)$ *in increasing order of* $\pi(w)$ **do**
3 $\quad \mathcal{N} \leftarrow \{v | v \in \text{nbr}(w, G') \wedge \pi(v) > \pi(w)\}$;
4 $\quad$ **for each** *pair of vertices* $u, v \in \mathcal{N}$ **do**
5 $\quad\quad$ **if** $(u, v) \notin E(G')$ **then**
6 $\quad\quad\quad$ insert $(u, v)$ into $G'$;
7 $\quad\quad\quad$ $\phi((u, v), G') \leftarrow \phi((u, w), G') + \phi((w, v), G')$;
8 $\quad\quad$ **else if** $\phi((u, w), G') + \phi((w, v), G') < \phi((u, v), G')$ **then**
9 $\quad\quad\quad$ $\phi((u, v), G') \leftarrow \phi((u, w), G') + \phi((w, v), G')$;
10 **for each** $w \in V(G)$ *in decreasing order of* $\pi(w)$ **do**
11 $\quad \mathcal{N} \leftarrow \{v | v \in \text{nbr}(w, G') \wedge \pi(v) > \pi(w)\}$;
12 $\quad$ **for each** *pair of* $u, v \in \mathcal{N}$ **do**
13 $\quad\quad$ **if** $\phi((w, v), G') + \phi((v, u), G') < \phi((w, u), G')$ **then**
14 $\quad\quad\quad$ $\phi((w, u), G') \leftarrow \phi((w, v), G') + \phi((v, u), G')$;
15 $\quad\quad\quad$ mark $(w, u)$ as removed;
16 remove all the marked edges in $G'$;
17 **for each** $v \in V(G')$ **do**
18 $\quad$ $\text{BNS}(v) \leftarrow \text{nbr}(v, G')$;

Based on Definition 5.3, we propose Algorithm 1 to compute the BN-Graph of an input road network and obtain the bridge neighbor set for each vertex accordingly. Intuitively, a BN-Graph of $G$ with larger bridge neighbor set for each vertex has more potential possibility to share the computation following the analysis of Section 5.1. Meanwhile, the construction of BN-Graph should not be costly. Following this idea, for a given road network $G$ and a total vertex order $\pi$ (the order used in our paper is discussed at the end of this section), our algorithm (Algorithm 1) contains two steps to construct BN-Graph: (1) Edge insertion, it aims to add edges to connect vertices to enlarge the bridge neighbor set. (2) Edge deletion, it deletes edges to guarantee that the bridge neighbor set is enlarged correctly. Specifically,

- *Step 1. Edge Insertion:* Given a graph $G$ and a rank over all vertices in $G$, it initializes $G'$ as $G$, and iterates every vertex in the increasing order of $\pi(w)$ (line 1-2). For every pair of vertices $u, v$ among the neighbors of $w$ in $G'$ with higher ranks than $w$, if $(u, v) \notin E(G')$, a new edge $(u, v)$ with weight $\phi((u, v), G') = \phi((u, w), G') + \phi((v, w), G')$ is inserted into $G'$ (line 5-7). Otherwise, if $\phi((u, w), G') + \phi((w, v), G') < \phi((u, v), G')$, it updates $\phi((u, v), G')$ as $\phi((u, w), G') + \phi((w, v), G')$ (line 8-9).

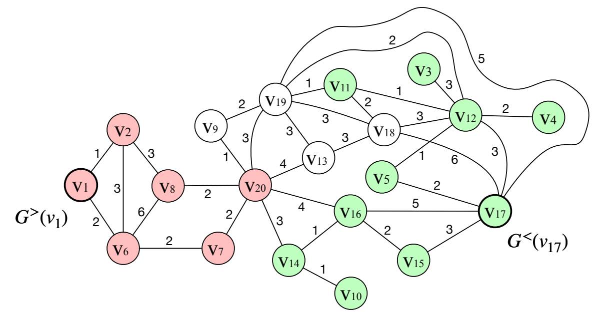



**Figure 4:** BN-Graph $G'$ **of** $G$

- *Step 2. Edge Deletion:* After the edge insertion step, it further iterates the vertex in the decreasing order of $\pi(w)$ (line 10). For every pair of vertices $u, v$ among the neighbors of $w$ in $G'$ with higher ranks than $w$ (line 11-12), if $\phi((w, v), G') + \phi((v, u), G') < \phi((w, u), G')$, it updates $\phi((w, u), G')$ as $\phi((w, v), G') + \phi((v, u), G')$ and marks the updated edge as removed (line 13-15). At last, the marked edges in $G'$ are removed (line 16), and $\text{BNS}(w)$ for each vertex is set as $\text{nbr}(w, G')$ (line 17-18).

*Example 5.4.* Consider the road network $G$ in Figure 2 and assume the vertex order $\pi = (v_1, v_2, ..., v_{20})$, the BN-Graph $G'$ of $G$ is shown in Figure 4. To construct $G'$, we first conduct the edge insertion step. For $v_1$, its $\mathcal{N}$ is $\{v_2, v_6\}$. There exists no edge $(v_2, v_6)$ in $G'$ currently, then $(v_2, v_6)$ with $\phi((v_2, v_6), G') = 3$ is added into $G'$. The procedure continues until all vertices are processed. In the edge deletion step, vertices are processed in the reverse order of $\pi$. Take $v_7$ as an example. When processing $v_7$, its $\mathcal{N}$ is $\{v_8, v_{20}\}$. Since $\phi((v_7, v_{20}), G') + \phi((v_{20}, v_8), G') = 2 + 2 < \phi((v_7, v_8), G') = 5$, $(v_7, v_8)$ is marked. When all the vertices are processed, the marked edges are removed, and Figure 4 shows the final $G'$.

Lemma 5.5. *The graph $G'$ generated at the end of Algorithm 1 is a* BN-Graph *of* $G$.

Following Lemma 5.5, it is clear that for each vertex $v \in V(G)$, its $k$NN in $G$ is the same as that in $G'$ based on the condition (3) of Definition 5.3. Moreover, $\text{nbr}(w, G')$ is the bridge neighbor set of $w$ in $G'$ based on the condition (2) of Definition 5.3. The following problem is how to compute $\mathsf{V_k}(u)$ for each vertex $u$ via $G'$ and $\text{BNS}(u)$. According to the discussion in Section 5.1, to fully utilize the intermediate computed results during the KNN-Index construction, we define a special type of path based on the given total vertex order as follows:

*Definition 5.6.* **(Monotonic Rank Path)** Given the BN-Graph $G'$ of a road network $G$, for a vertex $u \in V(G')$, a path $p(u, v) = (u = v_1, v_2, \ldots, v_j = v)$ in $G'$ is a decreasing rank path of $u$ if $\pi(v_j) < \pi(v_{j-1}) < \cdots < \pi(v_1)$, and it is an increasing rank path of $u$ if $\pi(v_j) > \pi(v_{j-1}) > \cdots > \pi(v_1)$.

*Definition 5.7.* **(Monotonic Rank Path Subgraph)** Given the BN-Graph $G'$ of a road network $G$, for a vertex $u$, the decreasing rank path subgraph of $u$, denoted by $\mathsf{G'^{<}}(u)$, is the subgraph induced by all decreasing rank paths of $u$ in $G'$. The increasing rank path subgraph, denoted by $\mathsf{G'^{>}}(u)$, is the subgraph induced by all increasing rank paths of $u$ in $G'$.

*Example 5.8.* Given the BN-Graph $G'$ in Figure 4, for vertex $v_1$, increasing rank paths of $v_1$ contain $p(v_1, v_2) = (v_1, v_2), p(v_1, v_6) = (v_1, v_6), p(v_1, v_8) = (v_1, v_2, v_8)$ or $(v_1, v_6, v_8), p(v_1, v_7) = (v_1, v_6, v_7)$ or $(v_1, v_2, v_6, v_7), p(v_1, v_{20}) = (v_1, v_2, v_8, v_{20}), (v_1, v_6, v_8, v_{20}), (v_1, v_2, v_7, v_{20}), (v_1, v_2, v_6, v_7, v_{20}), (v_1, v_2, v_6, v_8, v_{20})$. The increasing rank path subgraph of $v_1$, i.e., $\mathsf{G'^{>}}(v_1)$, is the subgraph induced by these paths, which is shown in pink in Figure 4. The decreasing rank path subgraph of $v_{17}$, i.e., $\mathsf{G'^{<}}(v_{17})$, can be obtained similarly, which is shown in green in Figure 4.

*Definition 5.9.* **(Decreasing Rank Partial *k*NN)** Given a vertex $u \in V(G)$ and a set of candidate objects $\mathcal{M}$, the decreasing rank partial *k*NN of $u$, denoted by $\mathsf{V}^{<}_{\mathsf{k}}(u)$, is the *k*NN of $u$ in $\mathsf{G'^{<}}(u)$.

Lemma 5.10. *Given a vertex $u \in V(G)$ in a road network $G$, $\mathsf{V_k}(u) \subseteq \cup_{w \in V(\mathsf{G'^{>}}(u))} \mathsf{V}^{<}_{\mathsf{k}}(w)$.*

Proof: This lemma can be proved directly following Property 1. □

Therefore, if we can obtain $\mathsf{V}^{<}_{\mathsf{k}}(w)$ for each vertex, we can obtain $\mathsf{V_k}(u)$ following Lemma 5.10. Moreover, we have:

Lemma 5.11. *Given a road network of $G$ and a set of candidate objects $\mathcal{M}$, let $u_1$ be the vertex with the lowest rank, we have $\mathsf{V}^{<}_{\mathsf{k}}(u_1) = \{\mathcal{M} \cap \{u_1\}\}$.*

Proof: From Definition 5.7, $V(\mathsf{G'^{<}}(u_1)) = \{u_1\}$. Based on Definition 5.9, $\mathsf{V}^{<}_{\mathsf{k}}(u_1) = \{\mathcal{M} \cap V(\mathsf{G'^{<}}(u_1))\} = \{\mathcal{M} \cap \{u_1\}\}$. The lemma holds. □

Based on Lemma 5.11, the decreasing rank partial *k*NN for the vertex with the lowest rank can be computed directly. Regarding the remaining vertices, we further divide $\mathsf{BNS}(u)$ into two parts: $\mathsf{BNS}^{<}(u)$ which contains the neighbors of $u$ in $G'$ with lower rank than $u$, i.e., $\mathsf{BNS}^{<}(u) = \{v | v \in \mathsf{BNS}(u) \land \pi(v) < \pi(u)\}$ and $\mathsf{BNS}^{>}(u)$ which contains the neighbors of $u$ in $G'$ with higher rank than $u$, i.e., $\mathsf{BNS}^{>}(u) = \{v | v \in \mathsf{BNS}(u) \land \pi(v) > \pi(u)\}$. We have:

Lemma 5.12. *Given a vertex $u \in V(G)$ in a road network $G$, $\mathsf{V}^{<}_{\mathsf{k}}(u) \subseteq \{\mathcal{M} \cap \{u\}\} \cup_{v \in \mathsf{BNS}^{<}(u)} \mathsf{V}^{<}_{\mathsf{k}}(v)$.*

Proof: This lemma can be proved directly based on Property 1 and Definition 5.9. □

Lemma 5.12 indicates the scope of $\mathsf{V}^{<}_{\mathsf{k}}(u)$ for each vertex. To obtain $\mathsf{V}^{<}_{\mathsf{k}}(u)$, we only need to compute the distance between $u$ and $w \in \{\mathcal{M} \cap \{u\}\} \cup_{v \in \mathsf{BNS}^{<}(u)} \mathsf{V}^{<}_{\mathsf{k}}(v)$, and retrieve the top $k$ objects. To avoid the expensive Dijkstra's algorithm, we define:

*Definition 5.13.* **(Decreasing Rank Shortest Path)** Given the BN-Graph $G'$ of a road network $G$, for two vertices $u, v \in V(G')$, the decreasing rank shortest path between $u$ and $v$ is the rank decreasing path from $u$ to $v$ with the smallest length in $G'$.

In BN-Graph $G'$ of $G$, for any two vertices $u, v \in G'$, one shortest path between $u$ and $v$ is a decreasing rank shortest path. We call the length of decreasing rank shortest path between $u$ and $v$ as decreasing rank distance and denote it as $\mathsf{dist}_{<}(u, v)$. We have:

Lemma 5.14. *Given the BN-Graph $G'$ of a road network $G$, for a vertex $u \in V(G')$, $\mathsf{dist}_{<}(u, v) = \min_{w \in \mathsf{BNS}^{<}(u)} \{\phi((u, w), G') + \mathsf{dist}_{<}(w, v)\}$, where $v \in \mathsf{V}^{<}_{\mathsf{k}}(u)$.*

Proof: Based on Definition 5.7 and Definition 5.9, we have $\mathsf{V}^{<}_{\mathsf{k}}(u) \subseteq \{\mathcal{M} \cap V(\mathsf{G'^{<}}(u))\}$. According to Definition 5.13, for $\forall v \in \mathsf{G'^{<}}(u)$, there is one decreasing rank shortest path between $u$ and $v$, which passes through one vertex $w \in \mathsf{BNS}^{<}(u)$. Therefore, $\mathsf{dist}_{<}(u, v) = \min_{w \in \mathsf{BNS}^{<}(u)} \{\phi((u, w), G') + \mathsf{dist}_{<}(w, v)\}$. □

Lemma 5.15. *Given the BN-Graph $G'$ of a road network $G$, for a vertex $u$, let $v \in \mathsf{V}^{<}_{\mathsf{k}}(u) \cap \mathsf{V_k}(u)$, if $\mathsf{dist}_{<}(u, v) = \mathsf{dist}((u, v), G')$, there is a shortest path between $u$ and $v$ in $G'$, which is also a decreasing rank shortest path.*

Proof: According to Definition 5.7 and Definition 5.9, if $v \in \mathsf{V}^{<}_{\mathsf{k}}(u)$, we know $v \in V(\mathsf{G'^{<}}(u))$. Based on Definition 5.13, there is one decreasing rank shortest path between $u$ and $v$. When $\mathsf{dist}_{<}(u, v) = \mathsf{dist}((u, v), G')$, there is a shortest path between $u$ and $v$ in $G'$, which is also a decreasing shortest path. □

Based on Lemma 5.11, Lemma 5.12, and Lemma 5.14, to obtain $\mathsf{V}^{<}_{\mathsf{k}}(u)$ for each vertex, we can adopt a bottom-up strategy based on the increasing order of $\pi(u)$, and the computed distance for a lower rank vertex can be re-used to compute the distance for a higher rank vertex. However, $\mathsf{V}^{<}_{\mathsf{k}}(u)$ only contains the vertices $v \in \mathsf{V_k}(u)$ whose shortest paths to $u$ pass through $\mathsf{BNS}^{<}(u)$, the vertices $v \in \mathsf{V_k}(u)$ whose shortest paths to $u$ pass through $\mathsf{BNS}^{>}(u)$ does not considered. Unfortunately, these vertices cannot be obtained by only exploring the vertices in $\cup_{v \in \mathsf{BNS}^{>}(u)} \mathsf{V}^{<}_{\mathsf{k}}(v)$ in the similar way as discussed above since this approach only explores the vertices whose ranks are not higher than $\max_{v \in \mathsf{BNS}^{>}(u)} \pi(v)$. On the other hand, we have the following lemmas regarding the distance between $u$ and $v \in \mathsf{V_k}(u)$ based on Lemma 5.10:

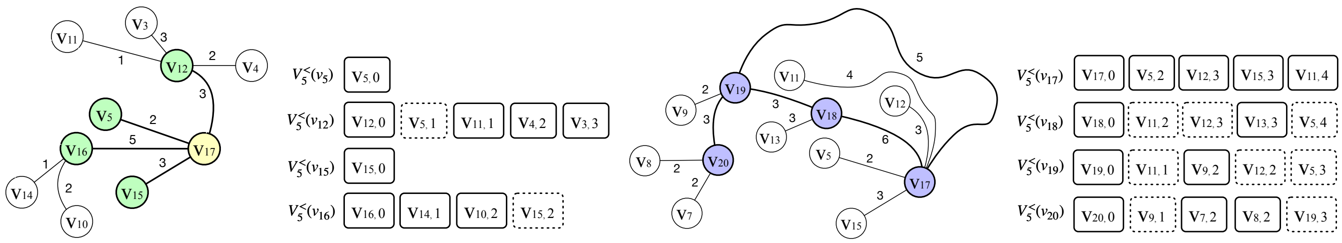
Lemma 5.16. *Given the BN-Graph $G'$ of a road network $G$, for a vertex $u \in V(G)$, let $v, v' \in V(\mathsf{G'^{>}}(u))$, $\mathsf{dist}((v, v'), \mathsf{G'^{>}}(u)) = \mathsf{dist}((v, v'), G)$.*

Proof: This lemma can be proved directly following Lemma 5.5. □

Lemma 5.17. *Given the BN-Graph $G'$ of a road network $G$, for a vertex $u \in V(G)$, $\mathsf{dist}((u, v), G) = \min_{w \in V(\mathsf{G'^{>}}(u))} \{\mathsf{dist}((u, w), \mathsf{G'^{>}}(u)) + \mathsf{dist}_{<}(w, v)\}$, where $v \in \mathsf{V_k}(u)$.*

Proof: This lemma can be proved directly based on Lemma 5.10 and Lemma 5.16. □

**Algorithm.** By combing the above two cases together, our index construction algorithm is shown in Algorithm 2. It first generates the BN-Graph $G'$ using Algorithm 1 (line 1). Then, it adopts a bottom-up strategy to compute $\mathsf{V}^{<}_{\mathsf{k}}(u)$ in the increasing order of $\pi(u)$ (line 2-7). Specifically, for each vertex $u$, it retrieves $\{\mathcal{M} \cap \{u\}\} \cup_{v \in \mathsf{BNS}^{<}(u)} \mathsf{V}^{<}_{\mathsf{k}}(v)$ based on Lemma 5.12 (line 4) and computes $\mathsf{dist}_{<}(u, v)$ based on Lemma 5.14 (line 5-6). Then, $\mathsf{V}^{<}_{\mathsf{k}}(u)$ is the $k$ vertices in $\mathcal{S}$ with the smallest $\mathsf{dist}_{<}(u, v)$ (line 7). After that, it constructs $\mathsf{G'^{>}}(u)$ by conducting BFS search from $u$ on $G'$ (line 9). And we compute the single source shortest distance $\mathsf{dist}((u, w), \mathsf{G'^{>}}(u))$ from $u$ to each vertex $w$ in $\mathsf{G'^{>}}(u)$ using the Dijkstra's Algorithm (line 10-11). Then, following Lemma 5.10, it retrieves $\cup_{w \in V(\mathsf{G'^{>}}(u))} \mathsf{V}^{<}_{\mathsf{k}}(w)$ (line 12) and computes $\mathsf{dist}((u, v), G)$ based on Lemma 5.17 (line 13-14). $\mathsf{dist}_{<}(w, v)$ can be obtained from $\mathsf{V}^{<}_{\mathsf{k}}(w)$ directly. $\mathsf{dist}((u, w), \mathsf{G'^{>}}(u))$ can be computed (line 10-11) after the construction of $\mathsf{G'^{>}}(u)$ (line 9) following Lemma 5.16. At last, the $k$ vertices in $\mathcal{S}$ with the smallest $\mathsf{dist}((u, v), G)$ is returned as $\mathsf{V_k}(u)$ in line 15.




(a) $\mathcal{S}$ for $\mathsf{V}_5^<(v_{17})$ (b) Computation of $\mathsf{V}_5^<(v_{17})$ (c) $\mathcal{S}$ for $\mathsf{V}_5(v_{17})$ (d) Computation of $\mathsf{V}_5(v_{17})$

**Figure 5: Procedure of Algorithm 2 to Compute $\mathsf{V}_5(v_{17})$**

**Algorithm 2:** KNN-Index-Cons($G, \pi, \mathcal{M}$)

```
1  G' ← SD-Graph-Gen(G, π);
2  S ← ∅, V_k^<(·) ← ∅, V_k(·) ← ∅;
3  for each u in increasing order of π(u) do
4      S ← {M ∩ {u}} ∪_{w∈BNS^<(u)} V_k^<(w);
5      for each v ∈ S do
6          dist_<(u, v) ← min_{w∈BNS^<(u)} {φ((u, w), G') + dist_<(w, v)};
7      V_k^<(u) ← k vertices in S with the smallest dist_<(u, v);
8  for each u in increasing order of π(u) do
9      construct G'^>(u) by conducting BFS search from u on G'
         following edge (v, v') with π(v) < π(v');
10     for each w ∈ V(G'^>(u)) do
11         compute dist((u, w), G'^>(u));
12     S ← V_k^<(u) ∪_{w∈V(G'^>(u))} V_k^<(w);
13     for each v ∈ S do
14         dist((u, v), G) ←
             min_{w∈G'^>(u)} dist((u, w), G'^>(u))+dist_<(w, v);
15     V_k(u) ← k vertices in S with the smallest dist((u, v), G);
```

*Example 5.18.* Following the BN-Graph $G'$ in Figure 4, Figure 5 takes $v_{17}$ as an example to show the procedure of Algorithm 2 to compute $\mathsf{V}_5(v_{17})$. According to Algorithm 2, we compute $\mathsf{V}_5^<(v_{17})$ first. Based on $G'$, $\mathsf{BNS}^<(v_{17}) = \{v_5, v_{12}, v_{15}, v_{16}\}$, which is shown in green in Figure 5 (a). Following Algorithm 2, when computing $\mathsf{V}_5^<(v_{17})$, we already have $\mathsf{V}_5^<(v_5), \mathsf{V}_5^<(v_{15}), \mathsf{V}_5^<(v_{12})$ and $\mathsf{V}_5^<(v_{16})$, which is shown in Figure 5 (b). Consequently, following line 6 of Algorithm 2, we can achieve $\mathcal{S} = \{\mathcal{M} \cap \{v_{17}\}\} \cup_{w\in\mathsf{BNS}^<(v_{17})} \mathsf{V}_5^<(w) = \{(v_{17}, 0), (v_5, 2), (v_{12}, 3), (v_{15}, 3), (v_{11}, 4), (v_4, 5), (v_{16}, 5), (v_3, 6), (v_{14}, 6), (v_{10}, 7)\}$. Figure 5 (b) shows this set $\mathcal{S}$ for constructing $\mathsf{V}_5^<(v_{17})$. After sorting distance, we have $\mathsf{V}_5^<(v_{17}) = \{(v_{17}, 0), (v_5, 2), (v_{12}, 3), (v_{15}, 3), (v_{11}, 4)\}$.

Figure 5 (c) shows the $\mathsf{G}'^>(v_{17})$ in purple with bold lines. Using Dijkstra's Algorithm, we compute the distance from $v_{17}$ to each vertex in $\mathsf{G}'^>(v_{17})$. And $\mathsf{dist}((v_{17}, v_{18}), \mathsf{G}'^>(v_{17})) = 6$, $\mathsf{dist}((v_{17}, v_{19}), \mathsf{G}'^>(v_{17})) = 5$, and $\mathsf{dist}((v_{17}, v_{20}), \mathsf{G}'^>(v_{17})) = 8$. Following line 12 of Algorithm 2, when computing $\mathsf{V}_5(v_{17})$, we have $\mathsf{V}_5^<(v_{18}), \mathsf{V}_5^<(v_{19})$ and $\mathsf{V}_5^<(v_{20})$, which is shown in Figure 5 (d). Following line 14 of Algorithm 2, we achieve $\mathcal{S} = \mathsf{V}_5^<(v_{17}) \cup_{w\in\mathsf{G}'^>(v_{17})} \mathsf{V}_5^<(w) = \{(v_{17}, 0), (v_5, 2), (v_{12}, 3), (v_{15}, 3), (v_{11}, 4), (v_{19}, 5), (v_{18}, 6), (v_9, 7), (v_{20}, 8), (v_{13}, 9), (v_7, 10), (v_8, 10)\}$, which is shown in Figure 5 (c). Then, we select 5 nearest objects from $\mathcal{S}$ as the KNN-Index of $v_{17}$, namely, $\mathsf{V}_k(v_{17}) = \{(v_{17}, 0), (v_5, 2), (v_{12}, 3), (v_{15}, 3), (v_{11}, 4)\}$ .

The correctness of Algorithm 2 is straightforward following the above discussion. For the efficiency of the algorithm, we have:

Theorem 5.19. *The time complexity of Algorithm 2 is bounded by $O(n\cdot(\rho^2+\eta\cdot\tau\cdot log(\eta)+(\tau+\eta)\cdot k))$, where $\rho$ represents the maximum degree of vertices in the graph generated by Algorithm 1 when Step 1 finishes, $\eta = \max_{v\in V(G)}|\mathsf{G}'^>(v)|$ and $\tau = \max_{v\in V(G)}|\mathsf{BNS}^>(v)|$.*

Proof: Algorithm 1 requires $O(n\cdot\rho^2)$ time (line 1 of Algorithm 2). This is because in the for loop (line 2-9 of Algorithm 1), for each vertex $w$, line 4-9 of Algorithm 1 takes $O(\rho^2)$ time and the for loop terminates at $n$ iterations. Therefore, the edge insertion step (line 2-9 of Algorithm 1) requires $O(n\cdot\rho^2)$ time. Similarly, the edge deletion step requires $O(n\cdot\rho^2)$ (line 10-16 of Algorithm 1). Scanning all vertices to achieve $\mathsf{BNS}(\cdot)$ is bounded by $O(n\cdot\tau)$ (line 17-18 of Algorithm 1). Obviously, for $\forall u \in V(G)$, $\tau \le \rho$. Therefore, the time complexity of Algorithm 1 is $O(n\cdot(\rho^2+\tau)) = O(n\cdot\rho^2)$. In the for loop from line 3 to line 7 of Algorithm 2, line 4 of Algorithm 2 takes $O(\tau\cdot k)$, since each vertex $u$ is only explored by the vertex $w \in \mathsf{BNS}^>(u)$. At the same time with obtaining $\mathcal{S}$ (line 4 of Algorithm 2), line 5-7 of Algorithm 2 could be done. Therefore, $\mathsf{V}_k^<(\cdot)$ construction (line 3-7 of Algorithm 2) requires $O(n\cdot\tau\cdot k)$ time. In the for loop (line 8-15 of Algorithm 2), constructing $\mathsf{G}'^>(u)$ by conducting BFS search requires $O(\eta\cdot\tau)$ time (line 9 of Algorithm 2). Computing $\mathsf{dist}(u, v)$ for $\forall v \in V(\mathsf{G}'^>(u))$ via Dijkstra's algorithm (line 10-11 of Algorithm 2) consumes $O(\eta\cdot\tau\cdot log(\eta))$ time. Obtaining $\mathcal{S}$ and distance computation require $O(\eta\cdot k)$ (line 12-15 of Algorithm 2). Therefore, $\mathsf{V}_k(\cdot)$ construction (line 8-15 of Algorithm 2) requires $O(n\cdot(\eta\cdot\tau\cdot log(\eta)+(\tau+\eta)\cdot k))$. In summary, the time complexity of Algorithm 2 requires $O(n\cdot(\rho^2+\eta\cdot\tau\cdot log(\eta)+(\tau+\eta)\cdot k))$. □

**Remark.** Based on Theorem 5.19, we prefer the generated BN-Graph with smaller $\rho$ and $\tau$. Thus, we use the following heuristic total order $\pi$ in this paper: (1) The vertex with the minimum degree in $G$ has the lowest rank (the vertex with a smallest id has the lowest rank if more than one vertices have the minimum degree); (2) for two unprocessed vertices $u$ and $v$ in line 2 of Algorithm 1, $\pi(u) > \pi(v)$ if the number of unprocessed neighbors of $u$ is bigger than that of $v$ in $G'$. Note this order can be obtained incidentally in Algorithm 1, and does not affect the time complexity of Algorithm 1.

## 5.3 A Bidirectional Construction Algorithm

Algorithm 2 adopts a bottom-up strategy to construct the KNN-Index with which the computation regarding $\mathsf{V}_k^<(u)$ is well shared. However, it still needs to invoke Dijkstra's algorithm to compute the distance between $u$ and $w \in V(\mathsf{G}'^>(u))$ in line 11-12, which

**Algorithm 3:** KNN-Index-Cons$^+(G, \pi, \mathcal{M})$

1 $G' \leftarrow$ SD-Graph-Gen$(G, \pi)$ ;
2 $\mathsf{V}^<_\mathsf{k}(\cdot) \leftarrow$ line 3-7 of Algorithm 2;
3 $\mathcal{S} \leftarrow \emptyset$, $\mathsf{V}_\mathsf{k}(\cdot) \leftarrow \emptyset$;
4 **for** ***each*** $u$ *in decreasing order of* $\pi(u)$ **do**
5 $\quad \mathcal{S} \leftarrow \mathsf{V}^<_\mathsf{k}(u) \cup_{w \in \mathsf{BNS}^>(u)} \mathsf{V}_\mathsf{k}(w)$;
6 $\quad$ **for** ***each*** $v \in \mathcal{S}$ **do**
7 $\quad\quad d \leftarrow \min_{w \in \mathsf{BNS}^>(u)} \{\phi((u, w), G') + \mathsf{dist}((w, v), G)\}$;
8 $\quad\quad \mathsf{dist}(u, v) \leftarrow \min\{d, \mathsf{dist}_<(u, v)\}$;
9 $\quad \mathsf{V}_\mathsf{k}(u) \leftarrow k$ vertices in $\mathcal{S}$ with the smallest $\mathsf{dist}((u, v), G)$;

is costly. To address this problem, we propose a new algorithm to further improve the index construction efficiency. Instead of following the sole bottom-up direction which adopted in Algorithm 2, the new algorithm constructs the index in a bidirectional manner, which totally avoids the invocation of Dijkstra′s algorithm. Before introducing our algorithm, we have:

LEMMA 5.20. *Given a road network $G$, let $u_n$ be the vertex with the highest rank,* $\mathsf{V}_\mathsf{k}(u_n) = \mathsf{V}^<_\mathsf{k}(u_n)$.

PROOF: Following Definition 5.7, $V(\mathsf{G}'^>(u_n)) = \{u_n\}$. Based on Lemma 5.10, $\mathsf{V}_\mathsf{k}(u_n) \subseteq \cup_{w \in V(\mathsf{G}'^>(u_n))} \mathsf{V}^<_\mathsf{k}(w) = \mathsf{V}^<_\mathsf{k}(u_n)$. □

LEMMA 5.21. *Given the* BN-Graph $G'$ *of a road network* $G$, *for a vertex* $u \in V(G)$, $\mathsf{V}_\mathsf{k}(u) \subseteq \mathsf{V}^<_\mathsf{k}(u) \cup_{w \in \mathsf{BNS}^>(u)} \mathsf{V}_\mathsf{k}(w)$.

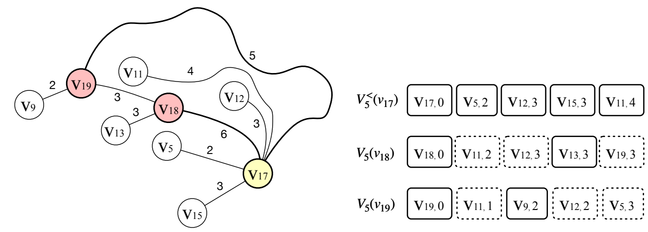
PROOF: This lemma can be proved directly based on Property 1. □

Lemma 5.20 and Lemma 5.21 imply that if we process the vertices in the decreasing order of their ranks when computing $\mathsf{V}_\mathsf{k}(u)$, it can re-use the computed information of vertices with higher ranks in the computation of the $k$NN for vertices with lower ranks. Moreover, we have:

LEMMA 5.22. *Given the* BN-Graph $G'$ *of a road network* $G$, *for a vertex* $u \in V(G)$, $\mathsf{dist}((u, v), G) = \min\{\min_{w \in \mathsf{BNS}^>(u)} \{\phi((u, w), G') + \mathsf{dist}((w, v), G)\}, \mathsf{dist}_<(u, v)\}$, *where* $v \in \mathsf{V}_\mathsf{k}(u)$.

PROOF: For $v \in \mathsf{V}_\mathsf{k}(u)$, there are two parts. The one part contains all $v$ whose shortest paths to $u$ pass through $\mathsf{BNS}^>(u)$, this distance computation can be proved based on Property 2. The other part contains all $v$ whose shortest paths to $u$ pass through $\mathsf{BNS}^<(u)$, $\mathsf{dist}_<(u, v)$ can be directly obtained from $\mathsf{V}^<_\mathsf{k}(u)$ based on Lemma 5.15. □

**Algorithm.** Following Lemma 5.22, our new bidirectional construction algorithm is shown in Algorithm 3. It first generates the BN-Graph $G'$ of $G$ using Algorithm 1 (line 1) and computes $\mathsf{V}^<_\mathsf{k}(u)$ in the same way as Algorithm 2 (line 2). After that, it processes the vertices in the decreasing order of their ranks (line 4-9). For each vertex $u$, it retrieves $\mathsf{V}^<_\mathsf{k}(u) \cup_{w \in \mathsf{BNS}^>(u)} \mathsf{V}_\mathsf{k}(w)$ based on Lemma 5.21 and stores them in $\mathcal{S}$ (line 5). Then, the distance between $u$ and $v \in \mathcal{S}$ is computed following Lemma 5.22 (line 6-8). Since the index construction procedure follows the decreasing order of $\pi(u)$, $\mathsf{V}_\mathsf{k}(w)$ for $\forall w \in \mathsf{BNS}^>(u)$ has been computed before computing $\mathsf{V}_\mathsf{k}(u)$. $\mathsf{dist}((w, v), G)$ can be obtained from $\mathsf{V}_\mathsf{k}(w)$ directly. And $\phi((u, w), G')$ can be achieved from BN-Graph directly. At last, the $k$ vertices in $\mathcal{S}$ with the smallest $\mathsf{dist}((u, v), G)$ is returned as $\mathsf{V}_\mathsf{k}(u)$ in line 9.




(a) $\mathcal{S}$ for $\mathsf{V}_5(v_{17})$ (b) Computation of $\mathsf{V}_5(v_{17})$

**Figure 6: Procedure of Algorithm 3 to Compute** $\mathsf{V}_5(v_{17})$

*Example 5.23.* Figure 6 shows the $\mathsf{V}_5(v_{17})$ construction procedure following Algorithm 3. Based on the BN-Graph $G'$ in Figure 4, for $v_{17}$, $\mathsf{BNS}^>(v_{17}) = \{v_{18}, v_{19}\}$, which is shown in pink in Figure 6 (a). $\mathsf{V}^<_5(v_{17})$ can be constructed in the same way as shown in Example 5.18. Following line 5 of Algorithm 3, when computing $\mathsf{V}^<_5(v_{17})$, we already have $\mathsf{V}^<_5(v_{17})$, $\mathsf{V}_5(v_{18})$, and $\mathsf{V}_5(v_{19})$, which is shown in Figure 6 (b). According to line 7-8 of Algorithm 3, we have $\mathcal{S} = \mathsf{V}^<_5(v_{17}) \cup_{w \in \mathsf{BNS}^>(v_{17})} \mathsf{V}_5(w) = \{(v_{17}, 0), (v_5, 2), (v_{12}, 3), (v_{15}, 3), (v_{11}, 4), (v_{19}, 5), (v_{18}, 6), (v_9, 8), (v_{13}, 9)\}$. After sorting distance, the 5 nearest neighbors for $v_{17}$ is selected from the set $\mathcal{S}$, namely, $\mathsf{V}_5(v_{17}) = \{(v_{17}, 0), (v_5, 2), (v_{12}, 3), (v_{15}, 3), (v_{11}, 4)\}$.

THEOREM 5.24. *Given a road network $G$, the time complexity of Algorithm 3 is bounded by $O(n \cdot \rho^2 + n \cdot \tau \cdot k)$ where $\rho$ represents the maximum degree of vertices in the graph generated by Algorithm 1 when Step 1 finishes, and* $\tau = \max_{v \in V(G)} |\mathsf{BNS}^>(v)|$.

PROOF: As proved in Theorem 5.19, Algorithm 1 requires $O(n \cdot \rho^2)$ time (line 1 of Algorithm 3) and $\mathsf{V}^<_\mathsf{k}(\cdot)$ construction requires $O(n \cdot \tau \cdot k)$ (line 2 of Algorithm 3). In the for loop from line 4 to line 9 of Algorithm 3, obtaining $\mathcal{S}$ and distance computation require $O(\tau \cdot k)$ (line 5-9 of Algorithm 3) and the loop terminates in $n$ iterations. Therefore, the for loop takes $O(n \cdot \tau \cdot k)$ (line 4-9 of Algorithm 3). In summary, the bidirectional KNN-Index construction (Algorithm 3) requires $O(n \cdot \rho^2 + n \cdot \tau \cdot k)$. □

## 6 CANDIDATE OBJECT UPDATE

In some cases, the candidate objects $\mathcal{M}$ may be updated by inserting new objects or deleting existing objects. Straightforwardly, we can reconstruct the index from scratch by Algorithm 3 to handle the update. However, this approach is inefficient as the update of a candidate object may not affect the $k$NN results of all the vertices. In this section, we discuss how to maintain the KNN-Index incrementally when the candidate objects are updated.

Obviously, when a candidate object $u$ is inserted or deleted, the update of $u$ will not affect the $k$NN results of a vertex $v$ if $u$ and $v$ are far away from each other. Specifically, let $v_k$ be the vertex in $\mathsf{V}_\mathsf{k}(v)$ with the largest distance to $v$. If $\mathsf{dist}(u, v) > \mathsf{dist}(v, v_k)$, then $u$ cannot be in $\mathsf{V}_\mathsf{k}(v)$, which means deleting or inserting $u$ will not affect $\mathsf{V}_\mathsf{k}(v)$. Moreover, we have the following lemma based on Property 1 and Property 2:

LEMMA 6.1. *Given the* BN-Graph $G'$ *of the road network* $G$, *for a vertex* $v \in V(G)$), *when an object* $u$ *is inserted/deleted,* $\mathsf{V}_\mathsf{k}(v)$ *could be affected if and only if there exists at least one vertex* $w \in \mathsf{BNS}(v)$ *whose* $\mathsf{V}_\mathsf{k}(w)$ *changes due to the update of* $u$.

Proof: This lemma can be directly proved based on Property 1 and Property 2. □

Therefore, we can maintain the KNN-Index starting from the vertex of the updated object $u$. Based on the definition of $k$NN, it is clear that $\mathsf{V_k}(u)$ will be changed. Following Lemma 6.1, the change of $\mathsf{V_k}(u)$ will possibly lead to the change of $\mathsf{V_k}(v)$ where $v \in \mathsf{BNS}(u)$. Then, we check whether $\mathsf{V_k}(v)$ needs to be updated based on $\mathsf{dist}(u, v)$ and $\mathsf{dist}(v, v_k)$ as discussed above. We continue to repeat the above procedure recursively, and it is obvious that the KNN-Index is correctly maintained when no more vertices whose $k$NN results change. Based on the above idea, our algorithms to handle the candidate object insertion and deletion are shown in Algorithm 4 and Algorithm 5, respectively.

**Algorithm 4:** KNN-Index-Ins($G'$, $\mathsf{V_k}(\cdot)$, $u$)

1 $\mathsf{dist}[\cdot] \leftarrow +\infty$; $\mathcal{S} \leftarrow \{\emptyset\}$; $Q \leftarrow \emptyset$;
2 $\mathsf{dist}[u] \leftarrow 0$; $\mathcal{S} \leftarrow \{u\}$; $Q.push(u)$;
3 **while** $Q \neq \emptyset$ **do**
4     $w \leftarrow Q.\mathsf{pop}()$;
5     **for *each*** $v \in \mathsf{BNS}(w)$ **do**
6         $\mathsf{dist}[v] \leftarrow \min\{\mathsf{dist}[v], \mathsf{dist}[w] + \phi((w, v), G')\}$;
7         **if** $v \notin \mathcal{S} \wedge \mathsf{checkIns}(v, \mathsf{V_k}(v), \mathsf{dist}[v])$ **then**
8             $Q.\mathsf{push}(v)$; $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$;
9 **for *each*** $v \in \mathcal{S}$ **do**
10     remove $v_k$ from $\mathsf{V_k}(v)$; insert $u$ into $\mathsf{V_k}(v)$;
11 **Procedure** $\mathsf{checkIns}(v, \mathsf{V_k}(v), d)$
12     $v_k \leftarrow$ the vertex with the largest distance to $v$ in $\mathsf{V_k}(v)$;
13     **if** $\mathsf{dist}(v, v_k) \leq d$ **then return** False;
14     **else return** True;

**Object Insertion.** Algorithm 4 shows the algorithm for candidate object insertion. An array $\mathsf{dist}[\cdot]$ stores the distance between $u$ and other vertices, a set $\mathcal{S}$ stores vertices whose $k$NN results should be updated, and a queue $Q$ stores the vertices whose bridge neighbor sets should be checked (line 1). Then, it initializes $\mathsf{dist}[u]$ as 0 and adds $u$ in $\mathcal{S}$ and $Q$ (line 1). After that, it pops a vertex $w$ from $Q$ (line 4), and for each $v \in \mathsf{BNS}(w)$, it computes the distance between $u$ and $v$, which can be obtained based on the fact $\mathsf{dist}(u, v) = \min_{w' \in \mathsf{BNS}(v)}\{\mathsf{dist}(u, w') + \mathsf{dist}(w', v)\}$ (line 6). Note that instead of visting all $w' \in \mathsf{BNS}(v)$, only the vertices whose $\mathsf{V_k}(w)$ changes due to the update of $u$ need to be explored following Lemma 6.1, which is captured by $Q$. If $v$ is not in $\mathcal{S}$ and $\mathsf{dist}[v]$ is smaller than the distance between $v$ and $v_k$, where $v_k$ is the vertex with the largest distance to $v$ in $\mathsf{V_k}(v)$ which can be obtained directly based on KNN-Index, it adds $v$ into $Q$ and $\mathcal{S}$ (line 7-8). The procedure terminates when $Q$ becomes empty (line 3). At last, for each vertex $v \in \mathcal{S}$, it removes $v_k$ from $\mathsf{V_k}(v)$ and inserts $u$ into $\mathsf{V_k}(v)$ (line 9-10). When inserting $u$ into $\mathsf{V_k}(u)$, $\mathsf{dist}[v]$ is the shortest distance between $u$ and $v$ guaranteed by Lemma 6.1.

Theorem 6.2. *Given the* BN-Graph $G'$ *and its corresponding* KNN-Index *of a road network G, Algorithm 4 maintains the* KNN-Index *correctly when an object u is inserted.*

Proof: Algorithm 4 (line 5-6) can guarantee $\mathsf{dist}[v]$ for $\forall v \in \mathcal{S}$ is the distance between $u$ and $v$ before inserting $u$ to $\mathsf{V_k}(v)$ for $v \in \mathcal{S}$ (line 9-10). Even though when using $\mathsf{checkIns}(v, \mathsf{V_k}(v), d)$ (line 7), $d$ may not be the distance between $u$ and $v$ and $\mathsf{checkIns}(v, \mathsf{V_k}(v), d)$ returns True, the final result can not be affected. Since $\mathsf{dist}(u, v) \leq d$, checkIns returning True denotes that $d < \mathsf{dist}(v, v_k)$. Therefore, we have $\mathsf{dist}(u, v) < \mathsf{dist}(v, v_k)$. Overall, Algorithm 4 maintains the KNN-Index correctly when an object $u$ is inserted. □

Theorem 6.3. *Given the* BN-Graph $G'$ *and its corresponding* KNN-Index *of a road network G, when an object u is inserted, Algorithm 4 maintains the* KNN-Index *in* $O(\Delta \cdot \tau')$*, where* $\Delta = |\mathcal{S}|$ *and* $\tau' = \max_{v \in V(G)} |\mathsf{BNS}(v)|$.

Proof: The time complexity of $\mathsf{checkIns}(v, \mathsf{V_k}(v), d)$ (line 11-14) is $O(1)$. In the for loop (line 3-8), for each vertex $w$, line 5-8 requires $O(\tau')$ time and the loop terminates in at most $\Delta$ iterations. Therefore, the for loop (line 3-8) requires $O(\Delta \cdot \tau')$ time. In the for loop (line 9-10), removing $v_k$ from $\mathsf{V_k}(v)$ requires $O(1)$, inserting $u$ into $\mathsf{V_k}(v)$ in the correct position needs $O(k)$ and the loop stops in $\Delta$ iterations. Therefore, the for loop (line 9-10) requires $O(\Delta \cdot k)$ time. In summary, the overall time complexity of Algorithm 4 is $O(\Delta \cdot (\tau' + k) = O(\Delta \cdot \tau')$, since $k$ is not large in real applications as discussed in Section 1. □

**Algorithm 5:** KNN-Index-Del($G'$, $\mathsf{V_k}(\cdot)$, $u$)

1 $\mathsf{dist}[\cdot] \leftarrow +\infty$; $\mathcal{S} \leftarrow \{\emptyset\}$; $Q \leftarrow \emptyset$;
2 $\mathsf{dist}[u] \leftarrow 0$; $\mathcal{S} \leftarrow \{u\}$; $Q.push(u)$;
3 **while** $Q \neq \emptyset$ **do**
4     $w \leftarrow Q.\mathsf{pop}()$;
5     **for *each*** $v \in \mathsf{BNS}(w)$ **do**
6         $\mathsf{dist}[v] \leftarrow \min\{\mathsf{dist}[v], \mathsf{dist}[w] + \phi((w, v), G')\}$;
7         **if** $v \notin \mathcal{S} \wedge \mathsf{checkDel}(u, v, \mathsf{V_k}(v), \mathsf{dist}[v])$ **then**
8             $Q.\mathsf{push}(v)$; $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$;
9 **for *each*** $v \in \mathcal{S}$ *in decreasing order of* $\pi(v)$ **do**
10     $\mathsf{processDel}(v, \mathsf{BNS}(v), \mathsf{V_k}(\cdot))$; delete $u$ from $\mathsf{V_k}(v)$;
11 **Procedure** $\mathsf{checkDel}(u, v, \mathsf{V_k}(v), d)$
12     $v_k \leftarrow$ the vertex with the largest distance to $v$ in $\mathsf{V_k}(v)$;
13     **if** $\mathsf{dist}(v, v_k) < d \vee u \notin \mathsf{V_k}(v)$ **then return** False;
14     **else return** True;
15 **Procedure** $\mathsf{processDel}(v, \mathsf{BNS}(v), \mathsf{V_k}(\cdot))$
16     $\mathcal{S}' \leftarrow \{\cup_{w \in \mathsf{BNS}(v)} \mathsf{V_k}(w)\} \setminus \mathsf{V_k}(v)$;
17     $v' \leftarrow \mathrm{argmin}_{v' \in \mathcal{S}'} \mathsf{dist}(v', v)$;
18     insert $v'$ into $\mathsf{V_k}(v)$;

**Object Deletion.** Algorithm 5 shows the algorithm for candidate object deletion, which follows a similar framework as Algorithm 4. The main difference is in line 10. When the vertices whose $\mathsf{V_k}(v)$ need to be updated are determined, Algorithm 5 finds a new vertex $v'$ to replace $u$ in $\mathsf{V_k}(v)$ by procedure processDel and deletes $u$ from $\mathsf{V_k}(v)$. For procedure processDel, it is easy to know that $v'$ must be the vertex in $\{\cup_{w \in \mathsf{BNS}(v)} \mathsf{V_k}(w)\} \setminus \mathsf{V_k}(v)$ with the smallest distance to $v$ according to Property 1, thus it first retrieves such set of vertices, namely $\mathcal{S}'$ (line 16), and finds the vertex in $\mathcal{S}'$ with the smallest distance to $v$ (line 17, since the vertices are processed in decreasing order of $\pi(v)$, $\mathsf{dist}(v', v)$ can be obtained in the similar way as line 7-8 of Algorithm 3 following the same idea). At last, it inserted $v'$ into $\mathsf{V_k}(v)$ in line 18.

Theorem 6.4. *Given the* BN-Graph $G'$ *and its corresponding* KNN-Index *of a road network G, Algorithm 5 maintains the* KNN-Index *correctly when an object u is deleted.*

Proof: Algorithm 5 (line 5-6) can guarantee dist$[v]$ for $\forall v \in \mathcal{S}$ is the distance between $u$ and $v$, before processing each $v \in \mathcal{S}$ (line 9-10). Even though when using checkDel$(v, \mathsf{V}_k(v), d)$ (line 7), $d$ may not be the distance between $u$ and $v$ and checkDel$(v, \mathsf{V}_k(v), d)$ returns True, the final result can not be affected. Since dist$(u, v) \leq d$, checkDel returning True denotes that $d <$ dist$(v, v_k)$. Therefore, we have dist$(u, v) <$ dist$(v, v_k)$. Overall, Algorithm 5 maintains the KNN-Index correctly when an object $u$ is deleted. □

Theorem 6.5. *Given the* BN-Graph $G'$ *and its corresponding* KNN-Index *of a road network G, when an object u is deleted, Algorithm 5 maintains the* KNN-Index *in* $O(\Delta \cdot \tau' \cdot k)$, *where* $\Delta = |\mathcal{S}|$ *and* $\tau' = \max_{v \in V(G)} |\mathsf{BNS}(v)|$.

Proof: The time complexity of checkDel$(u, v, \mathsf{V}_k(v), d)$ (line 11-14) is $O(1)$. In the for loop (line 3-8), for each vertex $w$, line 5-8 requires $O(\tau')$ time and the loop terminates in at most $\Delta$ iterations. Therefore, the for loop (line 3-8) requires $O(\Delta \cdot \tau')$ time. For the procedure processDel$(v, \mathsf{BNS}(v), \mathsf{V}_k(\cdot)$ (line 15-18), retrieving $\mathcal{S}'$ from $\{\cup_{w \in \mathsf{BNS}(v)} \mathsf{V}_k(w)\} \setminus \mathsf{V}_k(v)$ needs $O(|\mathsf{BNS}(v)| \cdot |\mathsf{V}_k(w)|) = O(\tau' \cdot k)$ time (line 16). At the same time with retrieving $\mathcal{S}'$, the vertex $v'$ with the smallest distance from $\mathcal{S}'$ can be achieved (line 17). Line 18 requires $O(1)$ time, since dist$(v, v') \leq$ dist$(v, v_k)$ and $v'$ should be inserted into the end of $\mathsf{V}_k(v)$. Therefore, the time complexity of processDel$(v, \mathsf{BNS}(v), \mathsf{V}_k(\cdot)$ (line 15-18) is $O(\tau' \cdot k)$. In the for loop (line 9-10), deleting $u$ from $\mathsf{V}_k(v)$ requires $O(k)$, and the loop stops in $\Delta$ iterations. Therefore, the for loop (line 9-10) requires $O(\Delta \cdot \tau' \cdot k)$ time. In summary, the overall time complexity of Algorithm 4 is $O(\Delta \cdot \tau' \cdot (1 + k)) = O(\Delta \cdot \tau' \cdot k)$. □

## 7 EXPERIMENTS

In this section, we compare our algorithms with the state-of-the-art method. All experiments are conducted on a machine with an Intel Xeon CPU and 384 GB main memory running Linux.

**Table 1: Datasets in Experiments**

| Dataset | Name | $n$ | $m$ | $\eta$ | $\tau$ | $\rho$ |
|---|---|---|---|---|---|---|
| New York City | NY | 264,346 | 733,846 | 725 | 56 | 116 |
| San Francisco Bay Area | BAY | 321,270 | 800,172 | 388 | 45 | 100 |
| Colorado | COL | 435,666 | 1,057,066 | 524 | 65 | 122 |
| Florida | FLA | 1,070,376 | 2,712,798 | 556 | 49 | 85 |
| Northwest USA | NW | 1,207,945 | 2,840,208 | 619 | 49 | 119 |
| Northeast USA | NE | 1,524,453 | 3,897,636 | 1096 | 81 | 149 |
| California and Nevada | CAL | 1,890,815 | 4,657,742 | 795 | 93 | 204 |
| Great Lakes | LKS | 2,758,119 | 6,885,658 | 1674 | 124 | 327 |
| Eastern USA | EUS | 3,598,623 | 8,778,114 | 1089 | 102 | 233 |
| Western USA | WUS | 6,262,104 | 15,248,146 | 1356 | 128 | 276 |
| Central USA | CTR | 14,081,816 | 34,292,496 | 2811 | 234 | 531 |
| Full USA | USA | 23,947,347 | 58,333,344 | 3315 | 257 | 587 |

**Datasets.** We use twelve publicly available real road networks from DIMACS [1]. In each road network, vertices represent intersections between roads, edges correspond to roads or road segments, the weight of an edge is the physical distance between two vertices. Table 1 provides the details about these datasets. Table 1 also shows the value of $\eta$, $\tau$ and $\rho$ for each road network. Clearly, $\eta$, $\tau$ and $\rho$ are small in practice.

[1] http://users.diag.uniroma1.it/challenge9/download.shtml

**Algorithms.** We compare the following algorithms:

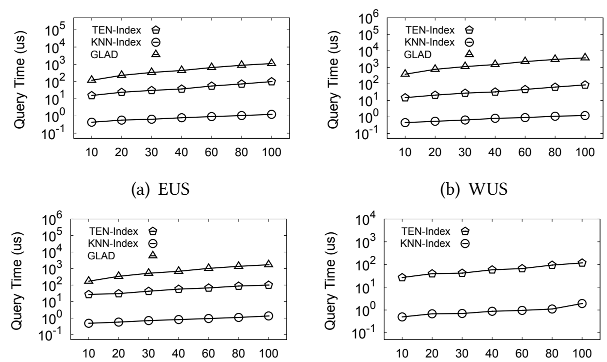
- TEN-Index: The state-of-the-art algorithm for $k$NN queries queries, which is introduced in Section 3.
- KNN-Index: Our proposed algorithms for $k$NN queries. For the index construction algorithms, we further distinguish between KNN-Index-Cons (Algorithm 2) and KNN-Index-Cons$^+$(Algorithm 3) for comparison.
- GLAD: Another algorithm for $k$NN queries proposed in [26], which is introduced in Section 8.
- Dijkstra-Cons: Using Dijkstra's Algorithm to compute top-$k$ nearest neighbors for all vertices in a given graph $G$ to construct the KNN-Index as discussed in Section 5.
- TEN-Index-Cons: Using TEN-Index to compute top-$k$ nearest neighbors for all vertices in a given graph $G$ to construct the KNN-Index as discussed in Section 5.

All the algorithms are implemented in C++ and compiled in GCC with -O3. The time cost is measured as the amount of wall-clock time elapsed during the program's execution. If an algorithm cannot finish in 6 hours, we denote the processing time as NA.

**Parameter Settings.** Following previous $k$NN works [17, 26, 33], we randomly select candidate objects in each dataset with a density $\mu = |\mathcal{M}|/|V|$. The candidate density $\mu$ and the query parameter $k$ settings are shown in Table 2, default values display in bold and italic font.

**Table 2: Parameter Settings**

| Parameters | Values |
|---|---|
| $\mu$ | 0.5, 0.1, 0.05, 0.01, ***0.005***, 0.001, 0.0005, 0.0001 |
| $k$ | 100, 80, 60, 40, 30, ***20***, 10 |




(a) EUS (b) WUS (c) CTR (d) USA

**Figure 7: Query Processing Time by Varying $k$**

**Exp-1: Query Processing Time when Varying $k$.** In this experiment, we evaluate the query processing time of our algorithms KNN-Index, the SOTA solutions TEN-Index and GLADby varying the parameter $k$. We randomly generate 10,000 queries and report average running time of each algorithm in Figure 7. Due to the limited space, only the results on four largest datasets are shown, and the remainings can be found in the technique report [51].

As shown in Figure 7, our algorithm is the most efficient one compared with TEN-Index and GLADand the growth for query processing time of TEN-Index and GLADis sharper than that of KNN-Index with increase of $k$. This is consistent with our analysis

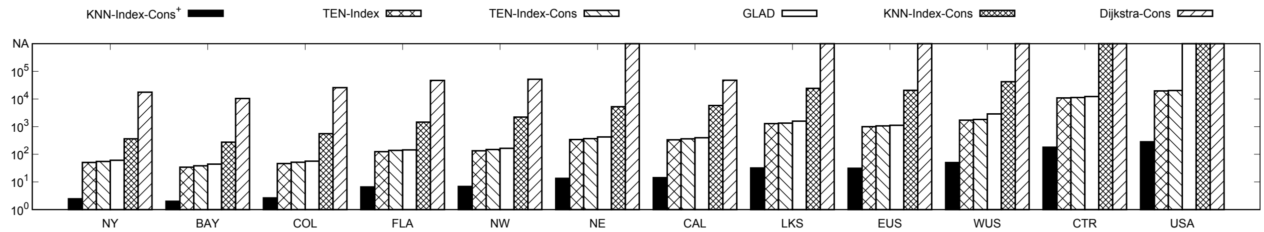
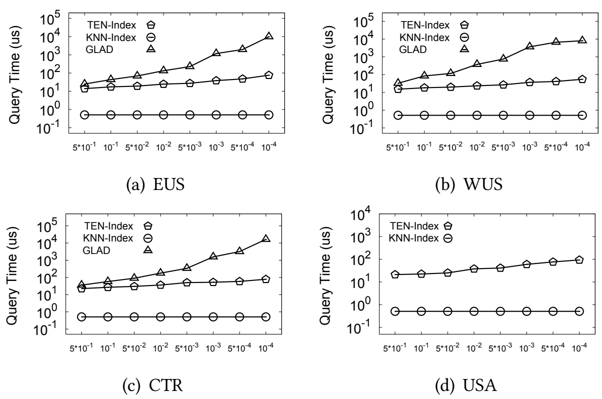



**Figure 8: Indexing Time (s)**

in Section 4.2. *For* GLAD*, it runs out of memory on* USA*, which means it is unable to handle large road networks. Thus, experimental results of* GLAD *on* USA *are not shown afterwards.*




(a) EUS (b) WUS (c) CTR (d) USA

**Figure 9: Query Processing Time by Varying $\mu = |\mathcal{M}|/|V|$**

**Exp-2: Query Processing Time when Varying $\mathcal{M}$.** We also compare our KNN-Index with the SOTA solutions TEN-Index and GLAD by varying object $\mathcal{M}$ (the density $\mu = |\mathcal{M}|/|V|$, therefore, we vary $\mathcal{M}$ by changing $\mu$). We randomly generate 10,000 queries for every dataset. We report the average processing time of each algorithm in Figure 9.

As shown in Figure 9, the query processing time of our algorithm is stable with the decrease of candidate object $\mathcal{M}$. However, the query processing time of TEN-Index and GLAD increases significantly with the decrease of candidate density $\mu$. For example, when $\mu = 0.0001$, KNN-Index achieves 2 orders of magnitude speedup compared with TEN-Index, and KNN-Index achieves up to 4 orders of magnitude speedup compared with GLAD. Moreover, the more sparsely the object set distributes, the larger speedup is. This is because our proposed algorithm is optimal regarding query processing as analyzed in Section 4.2.

**Exp-3: Indexing Time.** In this experiment, we evaluate the indexing time for KNN-Index-Cons$^+$, TEN-Index, TEN-Index-Cons, GLAD, KNN-Index-Cons and Dijkstra-Cons. Figure 8 shows that KNN-Index-Cons$^+$ is the fastest in all datasets, and achieves up to 2 orders of magnitude speedup compared with TEN-Index and GLAD. For example, KNN-Index-Cons$^+$ only takes 283.78s for USA while TEN-Index costs 19655.68s. TEN-Index-Cons and TEN-Index takes the similar indexing time as TEN-Index-Cons depends on the TEN-Index. They both rely on H2H-Index. Also, the indexing time of GLADand TEN-Index are similar, since GLADconstructs the additional grid index on the basis of H2H-Index. As shown in Figure 8, KNN-Index-Cons cannot complete the index construction within 6 hours for CTR and USA. And for USA GLADis out of memory. Dijkstra-Cons cannot finish index construction within 6 hours for NE, LKS, EUS, WUS, CTR and USA. Although index construction frameworks in KNN-Index-Cons and KNN-Index-Cons$^+$ are similar, KNN-Index-Cons consumes much more time compared with KNN-Index-Cons$^+$. For example, for WUS, KNN-Index-Cons$^+$ only costs 50.15s, but KNN-Index-Cons costs 42061.20s. This is because KNN-Index-Cons first uses BFS to construct $G'^{>}(u)$ for each vertex $u \in V(G)$, and then uses Dijkstra's Algorithm to compute $\text{dist}(u, v)$ for $\forall v \in V(G'^{>}(u))$ when constructing the index. However, KNN-Index-Cons$^+$ adopts a bidirectional construction strategy to avoid the time-consuming BFS search and the computation of Dijkstra's Algorithm during the index construction. The experimental results demonstrate the efficiency of our proposed algorithm regarding index construction.

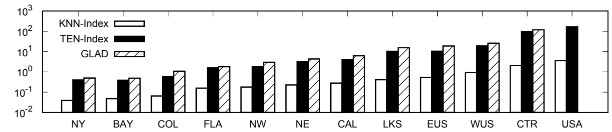



**Figure 10: Index Size (GB)**

**Exp-4: Index Size.** In this experiment, we evaluate the index size for KNN-Index, TEN-Index and GLAD. The experimental results for the 12 road networks are shown in Figure 10. Figure 10 shows the index size of KNN-Index is much smaller than that of TEN-Index and GLAD. For example, for the dataset USA, the KNN-Index size is only 3.57 GB while TEN-Index size is 169.28 GB, which is 47.42 times smaller than that of TEN-Index.

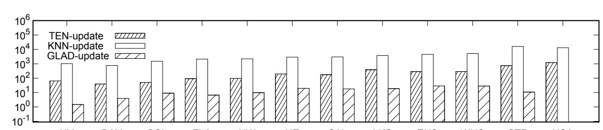



**Figure 11: Update Time (us)**

**Exp-5: Object Update.** In this experiment, we evaluate the performance of our update algorithms. To generate updated objects, we randomly select an object $u$ with either insertion or deletion. We skip the update if $u \notin \mathcal{M}$ for deletion and $u \in \mathcal{M}$ for insertion. For each dataset, we repeat this step until 10,000 updates are generated. The average time for each update is reported in Figure 11. The update time of KNN-Index is slower than that of TEN-Index and that of GLAD, since our update algorithm needs more time to compute the distance between each vertex and the updated objects. As analyzed in Section 3, TEN-Index contains H2H-Index, H2H-Index can compute the distance between any two vertices efficiently. Therefore, based on H2H-Index, TEN-Index can finish insertion or deletion in the shorter time. As GLAD only needs to update objects' grid index, the update performance is better than other approaches.

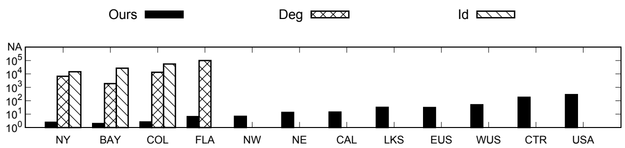


**Figure 12: Indexing Time of Different Vertex Orders (s)**

**Exp-6: Indexing Time of Different Vertex Total Orders.** We evaluate index construction performance using different total orders. We adopt three total orders: (1) degree-based total order in which the vertex with the smallest degree is processed first; (2) id-based total order in which the vertex with the smallest id is processed first; (3) and our proposed total order. As shown in Figure 12, the performance of our order is much better than others, which are consistent with our analysis in Section 5.2.

## 8 RELATED WORK

With the proliferation of graph applications, research efforts have been devoted to many problems in analyzing graph data [6, 56–58]. As a fundamental problem in graph data analysis, a direct approach to answer a $k$NN query is the Dijkstra's algorithm [15]. Nevertheless, this approach is inefficient obviously. Therefore, a plethora of index based Dijkstra-search enhanced solutions [11, 23, 26, 34, 40, 60] are proposed in the literature, which generally adopts the following search framework for a given query vertex $u$: (1) Initialize the distance for vertices $v$ it connected as their edge weights and other vertices as $+\infty$. (2) Maintain two vertex sets $S$ and $T$. $S$ contains vertices whose distance to $u$ is computed. $T$ contains vertices whose distance to $u$ is not computed yet, but have neighbors in $S$. Initially, $u$ is inserted in $S$ and the neighbors of $u$ are inserted in $T$. (3) Select one node $v$ with the smallest distance to $u$ from $T$, and add it to $S$. Then, the neighbors of $v$ are inserted into $T$. Here, different indexing methods add different restrictions, pruning unnecessary vertices to be inserted in $T$, to improve the query processing performance. (4) Repeat (3) until $|S| = k$.

Specifically, IER [34] uses Euclidean distance as a pruning bound to acquire the $k$NN results. INE [34] improves IER's Euclidean distance bound by expending searching space from the query location. [11] adapts a Euclidean restriction-based method to deal with continuous $k$ nearest neighbor problem. [11] divides the map into $N \times N$ grids and records which vertices and edges belong to some grid. Given a query vertex, the fixed distance between grids is used to filter a proximate range. ROAD [23] separates the input graph $G$ into many subgraphs hierarchically and skips the subgraphs without candidate objects to speedup $k$NN query processing. G-tree [60] adapts a binary tree division method to divide a graph into two disjoint subgraphs recursively until the number of vertices in a tree node is smaller than a predefined parameter. In each subgraph, G-tree maintains a distance matrix which stores distance between borders and vertices, which is used to prune unnecessary vertex exploration during the Dijkstra search. V-tree [40] constructs a similar structure as G-tree but adds additional $k$ nearest objects for borders, which leads to a faster query processing than G-tree. Based on the contraction hierarchy (CH) [16], TOAIN [26] constructs a $k$DNN index recording the top-$k$ nearest neighbors for each vertex $u$ from objects whose ranks are lower than $u$, where the rank is defined by the contraction hierarchy. To answer a $k$NN query with vertex $u$, TOAIN performs Dijkstra search from $u$ following the CH and maintains a candidate result set $R$. When visiting a vertex $v$, if there is a vertex $w$ in the $k$DNN of $v$ such that the distance of $w$ and $u$ is smaller than the $k$-th distance to $u$ in $R$, TOAIN updates $R$. The processing finishes when the Dijkstra search is far enough or all vertices are explored. Although the methods design different pruning algorithms to reduce the Dijkstra search space in step (3), the number of explored vertices cannot be well-bounded. In worst case, these methods degenerate into Dijkstra's algorithm, which leads to long query processing delay unavoidably. For TOAIN, asit constructs $k$DNN based on CH, which causes a relatively huge index size. Additionally, the vertex ranking method in TOAIN employs Dijkstra's Algorithm, which incurs an expensive time cost regarding index construction. The experimental results of [33] also verify above discussions.

Apart from the Dijkstra-search enhanced solutions, [24] exploits the massive parallelism of GPU to accelerate the $k$NN query processing. GLAD [17] partitions the road network into $2^x \times 2^x$ girds based on the geographical coordinate of each vertex. When answering a $k$NN query, it starts the search from the grid containing the query vertex and updates the candidate result via probing vertices in neighbor grids iteratively. It avoids the exploration to the vertices in a grid if the minimum Euclidean distance between any vertex inside the grid and the query vertex is not less than the largest distance in the candidate result. As GLAD needs to use H2H-Index to compute the exact shortest distance to select the final exact $k$NN results, the query processing is long. Moreover, since GLAD depends on H2H-Index, the index size of GLAD is huge and the indexing time of GLAD is long, which are similar to TEN-Index [33]. TEN-Index [33] is the state-of-the-art approach to $k$NN query in road network, which has been discussed in Section 3. [25] extend TEN-Index [33] and GLAD [17] onto time-dependent road networks. [19] extends tree decomposition method [32] to deal with $k$NN search on flow graph. Besides, continuous $k$NN query problem on road network is also studied in the literature [8, 9, 12, 19–22, 28, 39, 59]. Different from our setting, these studies generally assume that the query vertex is moving on the road network, and thus are orthogonal to ours.

## 9 CONCLUSION

Motivated by existing complex-index-based approaches for classical top $k$ nearest neighbors search in road networks suffers from the long query processing delay, oversized index space, and prohibitive indexing time, we embrace minimalism and design a simple index for $k$NN query. The index has a *well-bounded* space and supports progressive and optimal query processing. Moreover, we further design efficient algorithms to support the index construction. Experimental results demonstrate the significant superiority of our index over the state-of-the-art approach.

**Acknowledge.** Long Yuan is supported by National Key R&D Program of China 2022YFF0712100, NSFC62472225. Xuemin Lin is supported by NSFCU2241211 and U20B2046. Zi Chen is supported by NSFC6240071854, BK20241381, JSTJ-2023-XH055.

## REFERENCES


[1] Mohammad Reza Abbasifard, Bijan Ghahremani, and Hassan Naderi. 2014. A survey on nearest neighbor search methods. *International Journal of Computer Applications* 95, 25 (2014).

[2] Airnb. [n.d.]. *https://www.airbnb.com/*.

[3] Nitin Bhatia et al. 2010. Survey of nearest neighbor techniques. *arXiv preprint arXiv:1007.0085* (2010).

[4] Booking. [n.d.]. *https://www.booking.com/*.

[5] Y. H. Chang, Jia-Shung Wang, and Richard C. T. Lee. 1994. Generating All Maximal Independent Sets on Trees in Lexicographic Order. *Inf. Sci.* 76, 3-4 (1994), 279–296.

[6] Zi Chen, Long Yuan, Li Han, and Zhengping Qian. 2021. Higher-order truss decomposition in graphs. *IEEE Transactions on Knowledge and Data Engineering* 35, 4 (2021), 3966–3978.

[7] Alexander Chernev, Ulf Böckenholt, and Joseph Goodman. 2015. Choice overload: A conceptual review and meta-analysis. *Journal of Consumer Psychology* 25, 2 (2015), 333–358.

[8] Hyung-Ju Cho, Se Jin Kwon, and Tae-Sun Chung. 2013. A safe exit algorithm for continuous nearest neighbor monitoring in road networks. *Mob. Inf. Syst.* 9, 1 (2013), 37–53.

[9] Hyung-Ju Cho and Chin-Wan Chung. 2005. An efficient and scalable approach to CNN queries in a road network. In *PVLDB*, Vol. 2. 865–876.

[10] Deming Chu, Fan Zhang, Wenjie Zhang, Ying Zhang, and Xuemin Lin. 2024. Graph Summarization: Compactness Meets Efficiency. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[11] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. 2009. Efficient continuous nearest neighbor query in spatial networks using euclidean restriction. In *Proceedings of SSTD*. Springer, 25–43.

[12] Ugur Demiryurek, Farnoush Banaei Kashani, and Cyrus Shahabi. 2009. Efficient Continuous Nearest Neighbor Query in Spatial Networks Using Euclidean Restriction. In *Proceedings of SSTD*, Vol. 5644. 25–43.

[13] Dianping. [n.d.]. *https://www.dianping.com/*.

[14] DiDi. [n.d.]. *https://www.didiglobal.com.*

[15] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. Vol. 1. 269–271.

[16] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of WEA*. Springer, 319–333.

[17] Dan He, Sibo Wang, Xiaofang Zhou, and Reynold Cheng. 2019. An efficient framework for correctness-aware kNN queries on road networks. In *Proceedings of ICDE*. IEEE, 1298–1309.

[18] Xinlei Huang, Zhiqi Ma, Dian Meng, Yanran Liu, Shiwei Ruan, Qingqiang Sun, Xubin Zheng, and Ziyue Qiao. 2024. PRAGA: Prototype-aware Graph Adaptive Aggregation for Spatial Multi-modal Omics Analysis. *arXiv preprint arXiv:2409.12728* (2024).

[19] 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.

[20] 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 (2021), 1–18.

[21] Mohammad R. Kolahdouzan and Cyrus Shahabi. 2004. Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases. In *PVLDB*. 840–851.

[22] Mohammad R. Kolahdouzan and Cyrus Shahabi. 2005. Alternative Solutions for Continuous K Nearest Neighbor Queries in Spatial Network Databases. *GeoInformatica* 9, 4 (2005), 321–341.

[23] Ken CK Lee, Wang-Chien Lee, Baihua Zheng, and Yuan Tian. 2010. ROAD: A new spatial object search framework for road networks. *IEEE transactions on knowledge and data engineering* 24, 3 (2010), 547–560.

[24] 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 *Proceedings of ICDE*. 881–892.

[25] 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.

[26] 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. *PVLDB* 11, 5 (2018), 594–606.

[27] Lingkai Meng, Yu Shao, Long Yuan, Longbin Lai, Peng Cheng, Xue Li, Wenyuan Yu, Wenjie Zhang, Xuemin Lin, and Jingren Zhou. 2024. A survey of distributed graph algorithms on massive graphs. *Comput. Surveys* 57, 2 (2024), 1–39.

[28] Kyriakos Mouratidis, Man Lung Yiu, Dimitris Papadias, and Nikos Mamoulis. 2006. Continuous Nearest Neighbor Monitoring in Road Networks. In *PVLDB*. 43–54.

[29] Nikolaos Nodarakis, Angeliki Rapti, Spyros Sioutas, Athanasios K Tsakalidis, Dimitrios Tsolis, Giannis Tzimas, and Yannis Panagis. 2017. (A) kNN query processing on the cloud: a survey. In *Algorithmic Aspects of Cloud Computing: Second International Workshop, ALGOCLOUD 2016, Aarhus, Denmark, August 22, 2016, Revised Selected Papers 2*. Springer, 26–40.

[30] OpenRice. [n.d.]. *https://www.openrice.com/*.

[31] OpenTable. [n.d.]. *https://www.opentable.com.au/*.

[32] 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 SIGMOD*. 709–724.

[33] 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 SIGMOD*. 1781–1795.

[34] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. 2003. Query processing in spatial network databases. In *PVLDB*. Elsevier, 802–813.

[35] Frederick Muench Ph.D. 2010, November 1. The Burden of Choice. https://www.psychologytoday.com/ca/blog/more-tech-support/201011/the-burden-choice.

[36] Ziyue Qiao, Junren Xiao, Qingqiang Sun, Meng Xiao, and Hui Xiong. 2024. Holistic Memory Diversification for Incremental Learning in Growing Graphs. *arXiv preprint arXiv:2406.07413* (2024).

[37] Neil Robertson and Paul D Seymour. 1984. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B* 36, 1 (1984), 49–64.

[38] Barry Schwartz. 2004. The paradox of choice: Why more is less. *New York* (2004).

[39] Cyrus Shahabi, Mohammad R. Kolahdouzan, and Mehdi Sharifzadeh. 2003. A Road Network Embedding Technique for K-Nearest Neighbor Search in Moving Object Databases. *GeoInformatica* 7, 3 (2003), 255–273.

[40] 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 *Proceedings of ICDE*. IEEE, 609–620.

[41] Erose Sthapit. 2018. The more the merrier: Souvenir shopping, the absence of choice overload and preferred attributes. *Tourism management perspectives* 26 (2018), 126–134.

[42] Qingqiang Sun, Kai Wang, Wenjie Zhang, Peng Cheng, and Xuemin Lin. 2024. Interdependence-Adaptive Mutual Information Maximization for Graph Contrastive Learning. *IEEE Transactions on Knowledge and Data Engineering* (2024), 1–12.

[43] Trip. [n.d.]. *https://www.trip.com/*.

[44] Alina Tugend. 2010, February 27. Too many choices: A problem that can paralyze. https://www.nytimes.com/2010/02/27/your-money/27shortcuts.html?_r=1&.

[45] Uber. [n.d.]. *https://www.uber.com/*.

[46] Jianwei Wang, Kai Wang, Xuemin Lin, Wenjie Zhang, and Ying Zhang. 2024. Efficient Unsupervised Community Search with Pre-trained Graph Transformer. *PVLDB* 17, 9 (2024), 2227–2240.

[47] Jianwei Wang, Kai Wang, Xuemin Lin, Wenjie Zhang, and Ying Zhang. 2024. Neural Attributed Community Search at Billion Scale. *Proceedings of the ACM on Management of Data* 1, 4 (2024), 1–25.

[48] Jinghao Wang, Yanping Wu, Xiaoyang Wang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2024. Efficient Influence Minimization via Node Blocking. *PVLDB* 17, 10 (2024), 2501–2513.

[49] Jianwei Wang, Ying Zhang, Kai Wang, Xuemin Lin, and Wenjie Zhang. 2024. Missing Data Imputation with Uncertainty-Driven Network. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 117.

[50] Yiqi Wang, Long Yuan, Zi Chen, Wenjie Zhang, Xuemin Lin, and Qing Liu. 2023. Towards efficient shortest path counting on billion-scale graphs. In *Proceedings of ICDE*. IEEE, 2579–2592.

[51] Yiqi Wang, Long Yuan, Wenjie Zhang, Xuemin Lin, Zi Chen, and Qing Liu. 2024. Technical Report. https://arxiv.org/pdf/2408.05432.

[52] Yanping Wu, Renjie Sun, Xiaoyang Wang, Dong Wen, Ying Zhang, Lu Qin, and Xuemin Lin. 2024. Efficient Maximal Frequent Group Enumeration in Temporal Bipartite Graphs. *PVLDB* 17, 11 (2024), 3243–3255.

[53] Jiadong Xie, Zehua Chen, Deming Chu, Fan Zhang, Xuemin Lin, and Zhihong Tian. 2024. Influence Maximization via Vertex Countering. *PVLDB* 17, 6 (2024), 1297–1309.

[54] 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.

[55] Yelp. [n.d.]. *https://www.yelp.com/*.

[56] Long Yuan, Xia Li, Zi Chen, Xuemin Lin, Xiang Zhao, and Wenjie Zhang. 2024. I/O Efficient Label-Constrained Reachability Queries in Large Graphs. *PVLDB* 17, 10 (2024), 2590–2602.

[57] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. 2017. Index-based densest clique percolation community search in networks. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2017), 922–935.

[58] Junhua Zhang, Long Yuan, Wentao Li, Lu Qin, and Ying Zhang. 2021. Efficient Label-Constrained Shortest Path Queries on Road Networks: A Tree Decomposition Approach. *PVLDB* 15, 3 (2021), 686–698.

[59] 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 *Proceedings of ICDE*. IEEE Computer Society, 871–882.

[60] 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.