electronics

MDPI

Article

Graph-Indexed *k*NN Query Optimization on Road Network

Wei Jiang, Guanyu Li *, Mei Bai *, Bo Ning , Xite Wang and Fangliang Wei

Faculty of Information Science and Technology, Dalian Maritime University, Dalian 116026, China; merryaha@dlmu.edu.cn (W.J.); ningbo@dlmu.edu.cn (B.N.); xite-skywalker@163.com (X.W.); fangliangwei@139.com (F.W.)

* Correspondence: liguanyu@dlmu.edu.cn (G.L.); baimei@dlmu.edu.cn (M.B.)

Abstract: The nearest neighbors query problem on road networks constitutes a crucial aspect of location-oriented services and has useful practical implications; e.g., it can locate the *k*-nearest hotels. However, researches who study road networks still encounter obstacles due to the method's inherent limitations with respect to object mobility. More popular methods employ indexes to store intermediate results to improve querying time efficiency, but these other methods are often accompanied by high time costs. To balance the costs of time and space, a lightweight flow graph index is proposed to reduce the quantity of candidate nodes, and with this index the results of a *k*NN query can be efficiently obtained. Experiments on real road networks confirm the efficiency and accuracy of our optimized algorithm.

Keywords: *k*NN query; graph index; road network

1. Introduction

The nearest neighbors (top-*k* nearest neighbors (*k*NN)) query in a road network is an essential problem in location-based services. The *k*NN search is applied in various practical contexts. For example, when using hospitality reservation services such as Tongcheng-Elong, it is crucial to show a range of venues near the user's location. Similarly, in vehicle-for-hire platforms like Didi Chuxing, it is advantageous to offer a variety of vehicles within a short distance of the user.

Given a road network $G(V,E)$, a set of candidate objects M, and a query vertex q, the goal is to find the k objects in M that are closest to q. Figure 1 shows the example of a road network G, a moving object m_1 is on the edge from v_6 to v_3, $dist(v_3, v_6) = 2$, assuming all vertices are candidates. Given a query vertex v_3, the four nearest neighbors of v_3 are v_3, v_6, v_7, and v_{10}, with shortest network distances to v_3 of 0, 2, 3, and 4, respectively; then, the three nearest moving objects are m_1, m_2, and m_5.

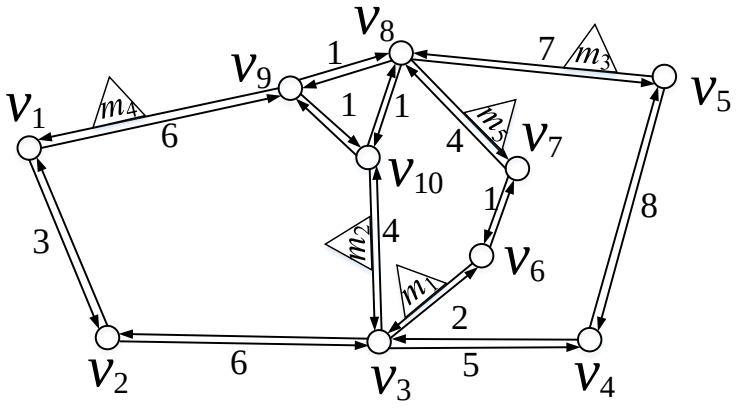


Figure 1. The example of a road network *G*.

check for updates

Citation: Jiang, W.; Li, G.; Bai, M.; Ning, B.; Wang, X.; Wei, F. Graph-Indexed *k*NN Query Optimization on Road Network. *Electronics* **2023**, *12*, 4536. https://doi.org/10.3390/electronics12214536

Academic Editor: Christos J. Bouras

Received: 7 October 2023
Revised: 28 October 2023
Accepted: 1 November 2023
Published: 3 November 2023

The kNN query problem on a road network has seen various research results. The most simple and direct solution is to use Dijkstra's algorithm [1]; i.e., given a query vertex q, search for vertices in order of non-decreasing distance from other vertices to q, ending when k vertices are found. Although this method is simple, the time cost is high, especially when the dataset is relatively large. The state-of-the-art algorithm for the kNN problem in road networks is TEN-Query [2], using the TEN-Index, which is based on tree decomposition. Mapping the graph into a tree structure, the kNN results for each vertex are precomputed and stored in the index. However, the establishment of the index is limited by the value of k, and the algorithm cannot perform queries when k increases. We have designed an index that breaks this limit by storing all candidate vertices in the local range of each vertex. Using a new tree decomposition generation algorithm, an average weight generation algorithm is used to construct tree decomposition.

The main contributions of this paper are as follows:

(1) Conventional tree decomposition is refined based on the minimum degree principle, and we propose an flow graph index generated from a graph-decomposed tree, which can effectively preserve distance information within the road network;
(2) Based on the flow graph index, the kNN query algorithm is proposed to significantly reduce the retrieval space of candidate nodes; thereby, it can enhance the performance of kNN queries in road networks;
(3) Experiments on real road networks confirm the efficiency and accuracy of the proposed algorithm.

The remainder of this paper is organized as follows. Section 2 reviews related literature. Section 3 describes the problem. Section 4 explains the index-construction methodology and query algorithm. Section 5 reviews our findings. Section 6 presents our conclusions.

2. Related Work

The kNN query can be based on Euclidean space or a transportation network. We concentrate on kNN queries within relaxation networks, which can be in either a static or dynamic environment.

The most elementary solution for kNN queries is Dijkstra's algorithm. Given a query point q, the algorithm searches for vertices in non-decreasing order of distance from the remaining vertices, stopping when k objects are found. Although effective, this is time-consuming, especially in large networks and when candidate objects are far away.

The grid-based index is a typical index structure in road network queries. D. He et al. [3] proposed a grid-based kNN index, GLAD, dividing road networks into $2^x \cdot 2^x$ grids according to the vertex latitude and longitude, maintaining a list of contained objects for each grid. To answer a kNN query, GLAD searches for objects starting from the mesh containing the queried vertices and updates the results by exploring adjacent meshes.

Dong Tianyang et al. [4] introduced direction-aware kNN (DAKNN) for moving objects on road networks, which employs an R-tree and a simple grid as fundamental structures to index the static road network and moving objects, respectively. An azimuth represents the moving direction of a target in the road network, and a local network extension method is used to quickly judge the direction of the moving target.

Graph partition-based indices are also used in kNN queries. Bilong Shen et al. [5] proposed a V-tree index to support efficient kNN searches and dynamic updates for moving objects. They iteratively divide the road network into sub-networks, build a tree structure on top of these, and associate a moving object with the nearest vertex in the V-tree. When updating the position of an object, only the tree nodes on the path from the corresponding leaf node to the root must be updated.

Zhong R et al. [6] proposed a balanced search tree index, G-tree, recursively dividing a road network into sub-networks, where each G-tree node corresponds to a sub-network. They introduced a best-first search algorithm on road networks, and an assembly-based approach to efficiently compute the minimum distance from a G-tree node to a query

location. Li Z et al. [7] proposed G*-tree, an improved *k*NN index structure, establishing shortcuts between selected leaf nodes to address the low query efficiency of G-tree.

Dan T et al. [8] proposed LG-tree, which partitions large graphs into subgraphs and indexes these using balanced trees. LG-tree constructs a hierarchical structure based on Dynamic Index Filtering (DIF) and boundary vertices, using stage-based dynamic programming to ensure the efficiency of the shortest distance search.

Bao JL [9] introduced the index structures used in nearest neighborhood queries, discussed current challenges, and explored future research focuses on nearest neighbor query techniques in road networks. Li L et al. [10] addressed the problem of continuously reporting alternative paths for users traveling along a given path by accessing vertices in ascending order of maximum depth value, so as to improve computational efficiency and enhance accuracy. Yuan-Ko Huang et al. [11] addressed the *k*NN skyline in dynamic road networks, conducting queries using three data structures: object attribute control matrix (OADM), road distance ranked list (RDSL), and skyline object extension tree (SOET). R. Halin et al. [12] introduced the concept of tree decomposition in 1976, expediting graph problem calculations by mapping graphs into tree structures. Dian Ouyang et al. [13] built on this with H2H-Query, a method to efficiently calculate the shortest distance between two vertices on a road network. The shortest distance for each vertex in each ancestor node is stored, enabling the most straightforward network distance calculation between any two points in $O(w)$ time. They subsequently proposed a tree-decomposition-based index, TEN-Index, which precalculates the shortest distance from each vertex to its ancestor vertex and stores each vertex's local *k*NN, which reduces query times and enhances *k*NN performance.

There are other *k*NN indices and algorithms. Hyung-Ju et al. [14] proposed an efficient continuous search algorithm, UNICONS, using Dijkstra's shortest path algorithm and a precalculated NN list to enhance NN query performance. Maytham Safar [15] used network distance rather than Euclidean distance as the shortest path metric. Guang Zhong Sun [16] suggested a straightforward, effective precomputation technique for *k*NN calculations on extensive road networks, selecting a suitable representative node set R (a subset of V) from road network $G(V,E)$ and precomputing the distance value for any pair in R.

In response to the natural extension of *k*NN queries for multiple moving objects, i.e., Aggregate *k* Nearest Neighbor (*Ak*NN) queries, Abeywickrama T [17] proposed the Compressed Object Landmark Tree (COLT) data structure, which implements efficient hierarchical graph traversal and performs various aggregation functions effectively. Bareche I et al. [18] proposed a Velocity Spatiotemporal (VeST) indexing approach for continuous queries, mainly Continuous *K*-Nearest Neighbor (*CK*NN) and continuous range queries using Apache Spark. The proposed structure is based on a selective velocity-partition method, which reduces the update cost and improves the response time and query accuracy. Ying Ju Chen et al. [19] proposed the SQUARE algorithm, constructing a network index resembling the decoupling model while using coupling concepts to maintain *k*NN information relative to hotspot areas in the network index. Hotspots represent areas where frequent queries are found in history. Aye Thida Hlaing et al. [20] proposed a fast *Rk*NN search algorithm using Simple Materialized Path View (SMPV), employing an Incremental Euclidean Restriction (IER) strategy for swift *k*NN queries. Luo et al. [21] proposed an index-based *k*NN query solution, Toain, based on a shrinkage hierarchy, and utilizing a shortcut-based index, SCOB, to accelerate query processing. Chuanwen Li et al. [22] proposed a kNN query algorithm using an index structure based on a lazy update strategy, capitalizing on both CPU and GPU resources. The region being queried is first identified, and the GPU is used to update the index for that region. The index is queried using the GPU and a candidate result set is generated, which is refined by the CPU to obtain the final result.

3. Problem Definitions

We study the *k*NN query problem on road networks based on tree decomposition, with symbols as defined in Table 1.

The road network is modeled as a weighted undirected graph $G(V,E)$, where V is a vertex set and E is an edge set. For an edge $e(v_1, v_2)$, weight $w(e)$ or $w(v_1, v_2)$ represents the length of the edge. A path p is a vertex sequence $p = (v_1, v_2, \ldots, v_k)$, where for any $1 \leq i < k$, $(v_i, v_{i+1}) \in E$, the weight of p is expressed as $\varphi(p) = \sum_{i=1}^{k-1} \varphi(v_i, v_{i+1})$ given two vertices v_1 and v_2; the shortest path between them is a path p from v_1 to v_2 that satisfies $w(p)$ as s minimum. In this paper, $\text{dist}(v_1, v_2)$ is used to represent the length of the shortest path from v_1 to v_2.

We study the road network kNN query in a static environment, i.e., when both the query point and candidate object are static. For the dynamic road network kNN query, the dynamic moving candidate object can be mapped into offset neighboring vertices; thus, it can be transformed into a static environment. Specifically, given a query point q and a moving object m, assuming that q is located on the edge $e_1 = (v_s, v_e)$, the distance from the vertex v_e is d_q, m is located on the edge $e_2 = (v'_s, v'_e)$, and the distance to the vertex v'_s is d_m. Then, the distance between q and m is expressed as $\text{dist}(q, m) = \text{dist}(v_e, v'_s) + d_q + d_m$. This paper defines the kNN query problem as follows.

Table 1. Symbol Definitions.

Symbol	Definition
$X(v)$.extList	Node expansion list of node $X(v)$, sorted in ascending order of road network distance from node $X(v)$
$v_{.isCandidate}$	Whether vertex v is a candidate vertex
$\text{dist}(v_1, v_2)$	The true shortest road network distance from vertex v_1 to vertex v_2
$\text{curDist}((v_1, v_2) \mid p)$	The current shortest road network distance from vertex v_1 to vertex v_2 along path p
R	kNN set of query point q, sorted in ascending order of road network distance from q
root	The root of the kNN index
threshold	Threshold
M	Candidate vertex set

Definition 1 (kNN Query). *Given a road network $G(V, E)$, a query point q, an integer k, and a set of candidate vertices M ($|M| > k$), the kNN query aims to calculate a set R that satisfies:*

1. $|R| = k$;
2. $R \subset M$;
3. $\forall v \in R, \forall v' \in M - R, dist(q, v) \leq dist(q, v')$

In the road network shown in Figure 2, there are ten vertexes, the distance between v_2 to v_3 is 6, given the query point v_3, with $k = 3$, and dotted line vertices (such as v_1, v_5, v_6, v_8, and v_{10}) are candidate vertices, the 3NN of v_3 in the candidate vertices is $R = \{v_6, v_{10}, v_8\}$, and their shortest network distances to v_3 are 2, 4, and 5, respectively.

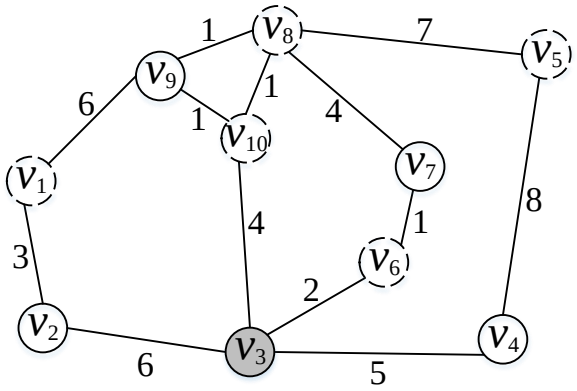


Figure 2. Example of kNN query of road network.

Notably, due to the potential for identical edge weights, the *k*NN for each vertex may need more uniqueness. This factor does not compromise the algorithm's validity, as presented in this treatise.

4. Index and Algorithm

We present the solution to the *k*NN problem on road networks, using an amalgamated index structure, Average Weight Tree Decomposition with Ancestor Table and Children Table (AWTDAC).

4.1. Average Weight Tree Decomposition Index Construction

Definition 2 (Tree Node)**.** *For each vertex v in the road network $G(V, E)$, its corresponding tree node is denoted by $X(v)$, $X(v)$ represents a subset of the road network vertex set V, comprising vertex v and its neighboring vertices. In this paper, v is referred to as the principal vertex of $X(v)$ and is denoted in gray, symbolized by $X(v)$.grayId. The neighboring vertices of v are considered adjacent vertices of $X(v)$.*

Taking the road network shown in Figure 1 as an example, Figure 3 depicts the tree node $X(v_3)$ corresponding to vertex v_3, where v_3 serves as the principal vertex, and v_1, v_4, and v_8 constitute adjacent vertices. The calculation process of the specific values of the adjacent vertices of v_3 is based on the average weight tree decomposition process. The result comes from the constructed decomposition tree.

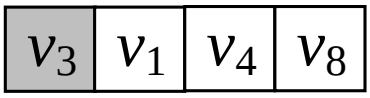


Figure 3. Example of tree node $X(v_3)$.

Tree decomposition [5] accelerates the computation of numerous graph problems by mapping graphs onto tree structures and is defined as follows.

Definition 3 (Tree Decomposition)**.** *Given a graph $G(V, E)$, its tree decomposition is denoted T_G, and is composed of several tree nodes with the set of all tree nodes represented as Λ. T_G satisfies the following conditions:*

(1) $\cup X(v) \in \Lambda X(v) = V$.
(2) *For each edge $e(v_1, v_2) \in E$ in G, there must exist a tree node $X(v_3) \in \Lambda$, satisfying $v_1 \in X(v_3)$ and $v_2 \in X(v_3)$.*
(3) *For each vertex v in G, all tree nodes $X(v')$ containing v (i.e., $v \in X(v')$) constitute a connected subtree of T_G.*

Definition 4 (Tree Height and Tree Width)**.** *Given a tree decomposition T_G, this manuscript stipulates that the root node's depth is 1, while the depth of other tree nodes is the depth of their parent node plus 1. Based on this, the tree height of T_G is defined as the maximum depth of all tree nodes, denoted by h. The treewidth of T_G is defined as the maximum number of adjacent vertices among all tree nodes, represented by w; that is,*

$$w = \max_{X(v) \in \Lambda} |X(v)| - 1 \tag{1}$$

Definition 5 (Elimination Order)**.** *Given a graph $G(V, E)$, this manuscript sequentially removes vertices in G following a specific rule. For each currently deleted vertex v, v.deleteOrder is utilized to record its removal order, referred to as the elimination order of vertex v.*

For instance, considering the road network $G(V, E)$ depicted in Figure 1, vertex v_6 is the first to be removed; hence, v_6.deleteOrder $= 1$.

We briefly describe the generation process of tree decomposition. For any road network $G(V, E)$, the traditional tree decomposition generation algorithm is based on the principle of minimum degree (i.e., the vertex with the smallest degree in graph G) and it is divided into two stages.

Stage One: Vertex Removal. Select vertex v with the smallest degree in G each time, and use v and its neighbors as tree nodes $X(v)$. For any pair of neighbors (v_1, v_2) of v, if no edge exists between v_1 and v_2, add an edge $e(v_1, v_2)$, whose weight $w(v_1, v_2) = w(v_1, v) + w(v, v_2)$; if there is an edge $e(v_1, v_2)$ and $w(v_1, v_2) > w(v_1, v) + w(v, v_2)$, then update $w(v_1, v_2)$ to $w(v_1, v_2) = w(v_1, v) + w(v, v_2)$. Finally, delete v from G. Repeat this step until all vertices in G have been removed and record the elimination order of each vertex.

Stage Two: Tree Decomposition Construction. For each tree node $X(v)$ generated in Stage One, select the vertex v' with the smallest elimination order among the adjacent vertices of $X(v)$, designating $X(v')$ as the parent node of $X(v)$. Repeat this step until all tree nodes coalesce into a tree.

Figure 4 exhibits a tree decomposition created using the minimum degree principle for the road network G illustrated in Figure 1. The gray vertices are the principal vertices, like v_1 is the principal vertex, and v_2, and v_9 constitute adjacent vertices. In this paper, a tree decomposition is constructed by the average weight principle. Both the average weight principle and the minimum degree principle are important concepts in graph algorithms, each having advantages in different types of problems. The average weight principle typically provides better results in dealing with weighted graph problems because it considers the weights of all nodes, not just their degrees. The minimum degree principle, on the other hand, is more suitable for specific problems such as connectivity issues. The average weight principle can provide improvements compared to the minimum degree principle, especially in dealing with weighted graph problems. However, the minimum degree principle may be more effective in dealing with certain specific types of problems. Therefore, the choice of which principle to use depends on the specific problem being addressed and the characteristics of the data.

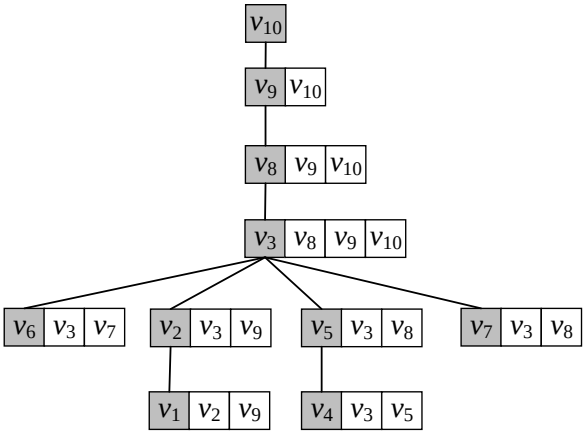


Figure 4. Tree decomposition T_G based on minimum degree.

The proposed average weight tree decomposition builds upon the minimum degree principle. The deletion of vertices no longer adheres to the minimum degree principle but defines a priority $S(v)(\alpha, \beta, \gamma)$ instead, where $S(v).\alpha$ represents the average weight of vertex v, i.e., the sum of edge weights of v divided by its degree. $S(v).\gamma$ is the number of edges that must be added after deleting vertex v, and $S(v).\gamma$ is the ID of vertex v. For the vertices in G, priority is sorted according to α; if α values are equal, sorting is based on β. Vertex v is removed after sorting $S(v)(\alpha, \beta, \gamma)$, and other operations remain unaltered. Figure 5 shows a tree decomposition generated based on the average weight principle for the road network G in Figure 1.

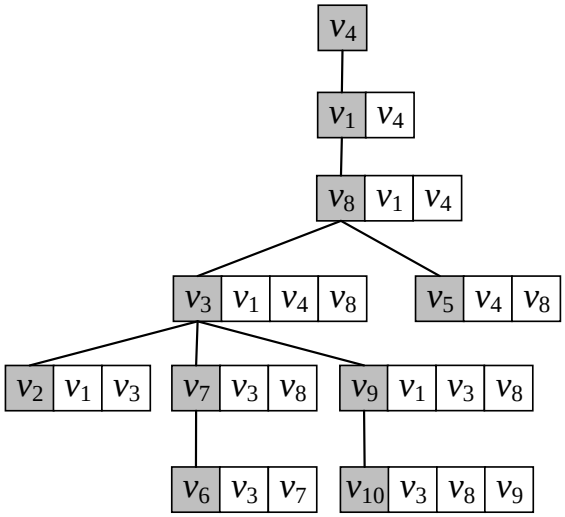


Figure 5. Tree decomposition T_G based on average weight.

Using Figure 5 as an example, the tree height of T_G is $h = 6$, and the tree width is $w = 3$, as the tree nodes within T_G contain a maximum of three adjacent vertices. For example, v_3 serves as the principal vertex, and v_1, v_4, and v_8 constitute adjacent vertices. In the road network, utilizing tree decomposition is reasonable, since the tree width and height are significantly less than the total number of vertices in the road network.

An average-weight-based tree decomposition algorithm is presented as Algorithm 1, where lines 2–9 represent vertex deletion, and lines 10–13 represent the tree decomposition phase. Upon the algorithm's completion, all tree nodes form a connected tree decomposition. The time complexity of Algorithm 1 is $O(n(w^2 + \log(n)))$, where n is the number of vertices in G, and w is the tree width of tree decomposition T_G. The space complexity is $O(nw)$.

Algorithm 1: Average Weight Tree Decomposition Construction Algorithm ($G(V, E)$).

Input: Road network $G(V, E)$
Output: Tree decomposition T_G

1 $T_G \leftarrow \emptyset$;
2 **for** $i = 1$ *to* $|V|$ **do**
3 Sort the vertices in G according to $S(v)(\alpha, \beta, \gamma)$;
4 $v \leftarrow$ Selected vertices after sorting ;
5 Create tree node $X(v) \leftarrow \{v\} \cup \{$neighbor vertices of $v\}$;
6 Adding edges to G makes all neighbors of v connected ;
7 Delete v and its neighbors from G ;
8 Update the α and β values of v's neighbors ;
9 v.deleteOrder $\leftarrow i$;
10 **end**
11 **for** $v \in V$ **do**
12 **if** $|X(v)| > 1$ **then**
13 $v' \leftarrow$ the vertex with the smallest elimination sequence in $X(v) - \{v\}$;
14 Set $X(v')$ as the parent node of $X(v)$ in T_G;
15 **end**
16 **end**
17 **Return** T_G

The calculated tree decomposition has the following properties [12].

Proposition 1. *For a graph G and its corresponding tree decomposition T_G, for any two distinct vertices, $v \neq v'$ in G, $X(v) \neq X(v')$.*

Proposition 2. *Considering a graph G and a tree decomposition T_G, for any tree node $X(v) \in \Lambda$ and any vertex $v' \in X(v)/v$, $X(v')$ is an ancestor of $X(v)$ in the tree node T_G.*

Definition 6 (Ancestor)**.** *With the tree decomposition TG of G, for any tree node $X(v) \in \Lambda$, the ancestor of $X(v)$ is expressed as the set $X(v).anc$, and the ancestor table of the root node is specified in this paper as $\emptyset$. Utilize $X(v).parent$ to represent the parent node of $X(v)$. Consequently, the ancestor $X(v).anc = X(v).parent.grayId \cup X(v).parent.anc$.*

Definition 7 (Children)**.** *Given the tree decomposition T_G of G, for any tree node $X(v)$, the child of $X(v)$ is symbolized as the set $X(v).child$. This paper stipulates that the leaf node's child is $\emptyset$, employing $X(v).childSet$ to represent the principal vertex set of $X(v)$'s child node and the child of the non-leaf node $X(v)$.*

$$X(v).child = \bigcup_{c \in X(v).childSet} c \cup X(c).child \tag{2}$$

Upon obtaining a tree decomposition, the shortest network distance between two vertices can be computed in $O(w)$ time using the H2H-Index [13], where w is the tree width. This is currently the most sophisticated technique to calculate the shortest net distance between vertices in road networks.

For any pair of vertices q and v with the nearest common ancestor X, the shortest path network distance between them is shown, which is described in Dian Ouyang et al. [13]:

$$dist(q, v) = \min_{s \in X} dist(q, s) + dist(s, v) \tag{3}$$

We introduce the proposed kNN index. Given a query point q, we traverse the ancestors and children of each vertex. For any other vertex v, there are three possible scenarios:

1. If $v \in X(q)$.anc, then dist(q, v) has been pre-calculated, and dist(q, v) can be obtained in $O(1)$ time.
2. If $v \in X(q)$.child, likewise, dist(q, v) can be obtained in $O(1)$ time.
3. If $v \notin X(q)$.child and $v \notin X(q)$.anc, according to Equation (3) , the shortest path network distance between v and q must pass through a vertex u in their nearest common ancestor. Hence, dist(q, v) = dist(q, u) + dist(u, v). This paper can obtain dist(q, u) and dist(u, v) in $O(1)$ time, allowing the retrieval of dist(q, v) within $O(1)$ time.

In summary, for any other vertex v, this paper can obtain dist(q, v) in $O(1)$ time.

Definition 8 (Node extension list)**.** *Given a tree decomposition T_G of G, for any tree node $X(v) \in \Lambda$, the node extension list of $X(v)$ is denoted as the set $X(v).extList$, where $X(v).extList = X(v)/v \cup X(v).child$.*

To enhance the algorithm's efficiency, this paper arranges the vertices in $X(v)$'s node extension list in ascending order based on their road network distance to vertex v.

For instance, in the tree decomposition T_G, illustrated in Figure 5, $X(v_3)$.extList = $\{(v_6, 2), (v_7, 3), (v_{10}, 4), (v_4, 5), (v_8, 5), (v_9, 5), (v_2, 6), (v_1, 9)\}$, with respective road network distances to v_3 of 2, 3, 4, 5, 5, 5, 6, and 9.

We were inspired by Sun et al. who put forward the idea of flow graph to serialize the processing graph [23]. According to the T_G structure on average weight tree decomposition in Figure 5, if the relationship between two vertices belongs to the parent–child relationship, the edge between the two vertices is called a tree edge; otherwise, it is a non-tree edge. The direction of the tree edge is from the parent vertex to the child vertex, and the direction of the non-tree edge is the direction from the deleted vertex to its adjacent vertices; in

Figure 6, the solid line indicates the tree edge, and the dashed line indicates the non-tree edge. It can be seen that v_3 is the parent vertices of v_2, v_7, v_8, and v_9, and the distances from child vertices to v_3 are 6, 5, 5, and 5. The adjacent vertices of v_3 are v_4, v_1, v_8.

Definition 9 (Flow Graph)**.** *Given a directed label graph $F(V_F, E_F, L_F)$, where V_F represents a set of vertices, E_F is a set of edges, and L_F represents a label function, assigning one or more labels to nodes in V_F. In this paper, L_F can be expressed as distance from two vertices, the vertex that contains the moving object M and the number of moving objects.*

Relying on average weight tree decomposition, we maintain a node extension list for each tree node, resulting in the *k*NN flow graph index structure depicted in Figure 6. If there is a parent-child relationship between two vertices, the edge between the two points is called a tree edge, otherwise it is called a non-tree edge. The direction of the tree edge is from the parent-child vertex to the child vertex. The direction of the non-tree edge is from the deleted vertex to its adjacent vertex. The solid line represents the tree edge and the dotted line represents the non-tree edge. The parent and child vertices of vertices v_2, v_7, v_8, and v_9 are v_3, and their distances to v_3 are 6, 5, 5, 5 respectively. The adjacent vertices of v_3 are v_4, v_1, v_8.

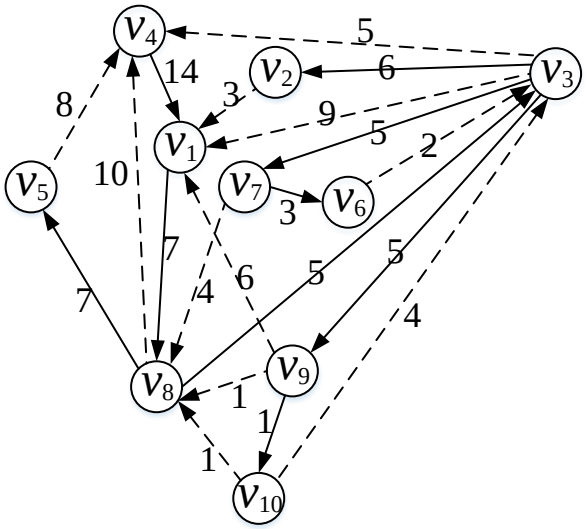


Figure 6. Example of *k*NN Flow Graph Index.

The space complexity of the *k*NN flow graph index is $O(n^2)$, where n is the total number of vertices in the road network. Using the flow graph index structure can completely preserve the index structure of the tree decomposition, and it can also save the weight between vertices and the information about the moving objects on the vertices.

Theorem 1 verifies the node completeness between the road network and the flow graph index.

Theorem 1. *Given a node v, if there exists a neighbor node v' of v, the tree node $X(v')$ is a passing node on the path from root node r to v in the graph-decomposed tree Λ rooted by node r.*

Proof of Theorem 1. Considering a node v and its neighbor nodes $N(v)$, node v can be abstracted as a tree node $X(v)$, satisfying $N(v) = X(v) - v$. If there exists a tree node $X(v')$, satisfying that $X(v')$ is a parent of $X(v)$, then $v' \in X(v) - v$. v' is connected with all other neighbor nodes $X(v) - v - v'$ after v is deleted. It is the same if there exists a tree node $X(v'')$, satisfying that $X(v'')$ is a parent of $X(v')$, such that $X(v'') \in (X(v) - v - v') \cup (X(v') - v')$. Similarly, all tree nodes abstracted from neighbor nodes of v can be found in a path from the root node to v. □

In addition, a filtering rule can be defined to prune the redundant nodes for kNN searching on the road network, shown as Lemma 1

Lemma 1. *Given a query node q and a data node v, if there exists a minimum common ancestor v' of q and v, the shortest path dist(q, v) is equal to the sum of shortest paths dist(q, v') and dist(v', v).*

Proof of Lemma 1. If there exists one other node v'', satisfying the condition that the shortest path dist(q, v) is equal to the sum of shortest paths dist(q, v'') and dist(v'', v), then there must exist one or more across-path edges to connect the node v''. However, it is impossible that there exists one across-path edge in a graph-decomposed tree according to Theorem 1 because any node v and its neighbor nodes are only found in the path from root node r to v. □

We present the index construction and update strategy. For each tree node in average weight tree decomposition, it is only necessary to compute its node extension list. The construction algorithm for the node extension list is shown as Algorithm 2, where lines 1–3 calculate the shortest network distance from adjacent vertices in $X(v)$ to v, adding it to the node extension list; lines 4–6 compute the child vertices of $X(v)$ and incorporate them into the node extension list; and line 7 sorts the node extension list. The time complexity of Algorithm 2 is $O(nw)$, where n is the total number of vertices in the road network, and w is the tree width of T_G.

Algorithm 2: Node Expansion List Construction Algorithm ($X(v)$)

Input: Tree node $X(V)$
Output: Node expansion list of $X(v)$
1 **for** *Each adjacent vertex p of* $X(v)$ **do**
2 Calculate dist(v,p);
3 $X(v)$.extList.push(p);
4 **end**
5 **for** *Each child vertex c of* $X(v)$ **do**
6 Calculate dist(v,c);
7 $X(v)$.extList.push(c);
8 **end**
9 sort($X(v)$.extList);
10 **Return** $X(v)$.extList

Regarding index updates, each vertex v has a flag bit, $v._{isCandidate}$, indicating its status as a candidate point, which is false when v ceases to be a candidate point. Consequently, index updates have time complexity $O(1)$.

4.2. Query-Processing Method Based on kNN Index

We examine the specifics of the query algorithm AWTDN.

Definition 10 (Ancestor Relationship)**.** *Given any two vertices v_1 and v_2, within the average weight tree decomposition T_G, if $v_1 \in X(v_2)$.child or $v_2 \in X(v_1)$.child, then v_1 and v_2 are said to possess an ancestral relationship.*

Definition 11 (Current Shortest Path Network Distance)**.** *Given vertices v_1 and v_2 with no ancestral relationship, and common ancestor $X(v_3)$, the current shortest network distance from vertex v_1 to vertex v_2 via vertex v_3 is represented as curDist$((v_1, v_2)|v_3)$, that is, curDist$((v_1, v_2)|v_3) =$ dist(v_1, v_3) + dist(v_3, v_2). Since $X(v_3)$ is the common ancestor of $X(v_1)$, $X(v_2)$, dist(v_1, v_3) and dist(v_3, v_2) can be found in $X(v_3)$.child without calculation.*

To enhance algorithm efficiency, this paper introduces a threshold for pruning, initially set to infinity and updated accordingly when the result is set $|R| \geq k$.

To facilitate the explanation of the kNN query problem, for a given query point q, we use a triplet (v_1, v_2, s) to suggest that v_1 via v_2 has the shortest network distance to query point q equal to s, with the intermediate point v_2 termed the base point.

The vertex expansion strategy of AWTDN has three steps.

Step 1: Insert the query point into the min-heap, process the first vertex p of the heap, and if p is a candidate point, add it to the result set R.

Step 2: Determine the relationship between p and the base point. (1) If p is a child of the base point, add the next vertex of p to the min-heap. (2) If p is an ancestor of the base point, add both the next vertex of p and the next vertex of the base point's node extension list to the min-heap. (3) If the distance between vertex v and the query point in the heap is updated, add the next vertex of vertex v to the heap.

Step 3: Process the heap's leading vertex, repeat the first two steps, and when the current road network distance from the heap's leading vertex to the query point is no less than the threshold, the algorithm concludes.

Consider a 3NN ($q = v_3$, $k = 3$) of the query vertex v_3 as an example. In the first step, v_3 is added to the min-heap, the first vertex p of the heap is processed, and the vertex $p = v_3$ is handled for the first time. As v_3 is not a candidate point, it is not processed. In the second step, with no ancestral or child relationship with the base point, the next vertex v_2 of v_3's node extension list is added to the min-heap. Correspondingly, in the third step, the heap's first vertex, v_2, is processed.

As mentioned earlier, this paper uses $\text{curDist}((q, v)|p) = \text{dist}(q, p) + \text{dist}(p, v)$ to denote the current shortest path network distance from vertex q to vertex v when passing through point p. Evidently, $\text{curDist}((q, v)|p)$ is the upper bound of $\text{dist}(q, v)$. Since $\text{dist}(q, p)$ and $\text{dist}(p, v)$ are pre-calculated, $\text{curDist}((q, v)|p)$ can be retrieved in constant time.

We present a lemma that supports the development of our query algorithm and two theorems that guide it and demonstrate its correctness.

Lemma 2. *For any given vertices q and v, if v is a kNN of q, there must exist a vertex $p \in X(q).extList \cup \{q\}$ such that v satisfies the following: (i) $v \in X(p).child$; (ii) $dist(q, v) = dist(q, p) + dist(p, v)$.*

As an illustration, consider Figure 6. Assume all vertices are candidate points, and given $q = v_3$ and $v = v_5$. For $\{v_3\} \in X(v_3)$.extList, we have $S = \{v_3, v_1, v_4, v_8\}$. There exists a vertex $p = v_8$ in S such that the shortest path from v_3 to v_5 passes through vertex v_8.

Based on Lemma 2, we propose our query algorithm. For a given query point q, we scan each vertex p in $\{q\} \cup X(q)$.extList. We first determine whether p is a candidate point. If so, we compare it with the last vertex in result set R, and update R accordingly. We then assess the relationship between p and the base point and process it accordingly.

Theorem 2. *For a query vertex q, vertex v, and their common ancestor p, if $curDist((q, v)|p) \geq$ threshold, then vertex v cannot be a kNN of q.*

Proof of Theorem 2. Given a query vertex q, assume that there is a vertex v and their common ancestor p, satisfying $\text{curDist}((q, v)|p) \geq$ threshold, and v is a kNN of q. Let R be the kNN set of q. It is easy to know that when threshold $\neq +\infty$, there is $|R| \geq k$, and for any point v' in R, there is $\text{dist}(q, v') \leq$ threshold, so there is $\text{dist}(q, v') \leq$ threshold $\leq \text{curDist}((q, v)|p)$, and from the definition of kNN, we can know that v is not the kNN of q, which contradicts the hypothesis. The proof is completed. □

Based on Theorem 1 and the ascending order feature of the node expansion list, when $\text{curDist}((q, v)|p) \geq$ threshold, then the vertex v and the vertices after v in the node expansion list must not be kNN of q, so we can exit the calculation early.

Theorem 3. *Assume that in* $\{q\} \cup X(q).extList$*, the algorithm is currently processing vertex* p*, and it terminates when* $dist(q, p) \geq threshold$*. At this point, for any vertex* v *in the result set* R*,* $curDist((q, v)|v') = dist(q, v)$ *holds, where* v' *is any common ancestor of* q *and* v*.*

Proof of Theorem 3. Let v be any vertex in the result set R. Given the termination condition, for any vertex p_{after} processed after p (based on the sequence of scanning in $\{q\} \cup X(q)$.extList), by the property of ascending order in node expansion lists, if $v \in X(p_{\text{after}})$.child, then we have $\text{dist}(q, p_{\text{after}}) + \text{dist}(p_{\text{after}}, v) \geq$ threshold. Considering that $\text{curDist}((q, v)|v') \leq$ threshold and v' is any common ancestor of q and v, $\text{curDist}((q, v)|v') \leq$ threshold; the proof is complete. □

The pseudocode of AWTDN is shown as Algorithm 3. For each vertex p in $\{q\} \cup X(q)$.extList, if the shortest network distance from p to q is not less than the threshold (line 5 of Algorithm 3), the algorithm terminates because no more kNNs of q will appear. Then, we evaluate p; if p is a candidate point, we update the result set R and the threshold.

Algorithm 3: kNN Searching Algorithm basd on Graph-Decomposed Tree.

Input: road network $G(V,E)$, query point q, integer k
Output: result set R
1 threshold $\sigma \leftarrow +\infty$, $R \leftarrow \emptyset$, small root heap $h \leftarrow \emptyset$;
2 $\Lambda(X(V), E(X(V))) \leftarrow$ construct graph-decomposed tree of $G(V,E)$, h.add(q);
3 **while** *h is not empty* **do**
4 The heap head vertex of $p \leftarrow h$;
5 **if** *dist(q, p) > threshold* σ **then**
6 break;
7 **end**
8 **if** $p_{.isCandidate} == true$ **then**
9 Update R and threshold σ;
10 **end**
11 Determine the relation between p and the base point;
12 Update h;
13 **end**
14 **Return** R

The space complexity of the graph-decomposed tree is $O(\Lambda) = \frac{|V|(|V|+1)}{2} + (|V| - 1)$ in the worst case, where each node can be connected by all other nodes, the number of tree edges is $|V| - 1$. The time complexity of the graph-decomposed kNN searching algorithm is $O(q) = |V| - 1$ for query node q, because the nearest nodes can be found in the tree nodes $X(q)$ on the graph-decomposed tree Λ.

For example, given a query point $q = v_3$, $k = 3$, suppose the candidate set $M = \{v_1, v_5, v_6, v_8, v_{10}\}$ (i.e., the vertices of dotted lines in Figure 2), and we query the 3NN of v_3. Table 2 shows the process. The meaning of triplet representation is as mentioned above. In rows 3–5 of Table 2, the relationship between v_6, v_7, v_{10}, v_9, v_2, and the base point v_3 is as follows. All are children of v_3, so we add the subsequent vertices in the extended list of v_3 nodes to the heap. In line 6, v_4 is the ancestor of base point v_3, so we must add the subsequent vertices in the extended lists of nodes v_3 and v_4 to the heap; note that at this time, the first vertex of the v_4 node expansion list is v_1, and $\text{dist}(v_4, v_1) = 14$, $\text{dist}(v_3, v_4) = 9$. Also, $\text{dist}(v_3, v_4) + \text{dist}(v_4, v_1) > \text{dist}(v_3, v_1)$. Hence, the shortest distance from v_4 to v_3 through v_1 is greater than the shortest distance from v_1 itself to v_3, so v_1 does no processing, and we add the follow-up vertex v_5 of v_4 to the heap. In the last line, the first vertex of the heap in the small root heap is v_2, and its label is (v_2,v_3,6); i.e., the shortest distance from v_2 to the query point after v_3 is 6, which is greater than the threshold of 5 at this time. So, the algorithm ends and returns R=(v_6,2),(v_{10},4),(v_8,5), indicating that the 3NN of v_3 is v_6, v_{10}, v_8, and their respective shortest distances to the query point are 2, 4, and 5.

This example helps illustrate the efficiency of AWTDN in handling *k*NN queries. The algorithm can use the precomputed shortest distances and a threshold value to prune unnecessary computations, improving overall performance.

By focusing on an average weight tree decomposition instead of on the minimum degree, we can more accurately capture the inherent structure of the road network. It visits fewer vertices for a given query point, which increases computational efficiency, as the number of visited vertices of a query point q is much smaller, as confirmed by our experiments, because average weight tree decomposition better preserves the distance information on the road network. On average, the vertices are similar to those in intermediate weight tree decomposition.

Table 2. *k*NN solution process of v_3 ($q = v_3$, $k = 3$).

Currently Processed Vertex	Candidate Point	Result Set	Relationship with Base Point	Extended Vertex Label	Min-Heap	Threshold
$(v_3, \emptyset, 0)$	No			$(v_6, v_3, 2)$	$(v_6, v_3, 2)$	$+\infty$
$(v_6, v_3, 2)$	Yes	$(v_6, 2)$	Child	$(v_7, v_3, 3)$	$(v_7, v_3, 3)$	$+\infty$
$(v_7, v_3, 3)$	No	$(v_6, v_3, 2)$	Child	$(v_{10}, v_3, 4)$	$(v_{10}, v_3, 4)$	$+\infty$
$(v_{10}, v_3, 4)$	Yes	$(v_6, 2)$,$(v_{10}, 4)$	Child	$(v_4, v_3, 5)$	$(v_4, v_3, 5)$	$+\infty$
$(v_4, v_3, 5)$	No	$(v_6, 2)$,$(v_{10}, 4)$	Ancestor	$(v_8, v_3, 5)$, $(v_1, v_4, 14)$	$(v_8, v_3, 5)$, $(v_1, v_4, 14)$	$+\infty$
$(v_8, v_3, 5)$	Yes	$(v_6, 2)$,$(v_{10}, 4)$, $(v_8, 5)$	Ancestor	$(v_9, v_3, 5)$, $(v_1, v_8, 14)$, $(v_5, v_8, 17)$	$(v_9, v_3, 5)$, $(v_1, v_8, 14)$, $(v_5, v_8, 17)$	$+\infty$
$(v_9, v_3, 5)$	No	$(v_6, 2)$,$(v_{10}, 4)$, $(v_8, 5)$	Child	$(v_2, v_3, 6)$, $(v_1, v_9, 14)$, $(v_5, v_9, 17)$	$(v_2, v_3, 6)$, $(v_1, v_9, 14)$, $(v_5, v_9, 17)$	5

4.3. kNN-Query-Processing Method in a Dynamic Environment

We focus on the *k*NN query problem in a dynamic road network environment, where the candidate points are moving objects. Specifically, given a query point q, we must find the k moving objects that are closest. Since the positions of moving objects change with time, we use the offset method to map them to the nearest road network vertex and then convert the query from a dynamic environment to a static environment.

Considering the directionality of moving objects, we model the road network as a directed weighted graph $G(V, E)$.

Definition 12 (Moving Object)**.** *Each moving object on the road network is represented by* $m = < (v_i, v_j), offset, t >$*; that is, at time* t*, moving object* m *is on edge* (v_i, v_j)*, at an offset from vertex* v_j*.*

For instance, referring to Figure 7, if the current time is t, moving object m_{13} can be represented as m_{13} = < (v_3,v_2),2,t>, meaning m_{13} is on the edge between v_3 and v_2, the distance between v_3 and v_2 is 5 and the offset from vertex v_2 is 2. The algorithm maps each query point to the starting vertex with the offset. For example, if q_1 is located on edge (v_2, v_1), it is mapped to starting point v_2 with an offset of 1. Therefore, the distance between a moving object $m =< (v_i, v_j), offset_1, t >$ and query point $q =< (v_s, v_t), offset_2, t >$ can be represented as $offset_1 + \text{dist}(v_j, v_s) + offset_2$.

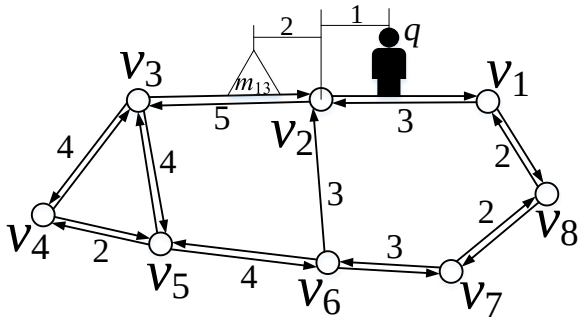
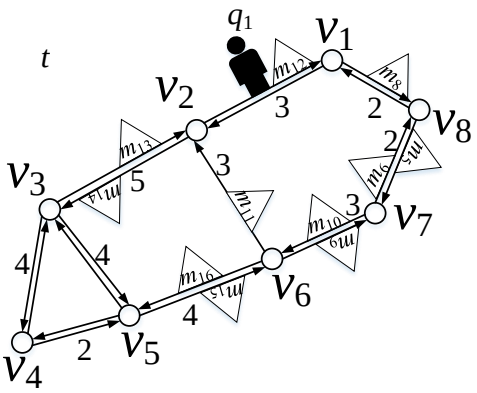
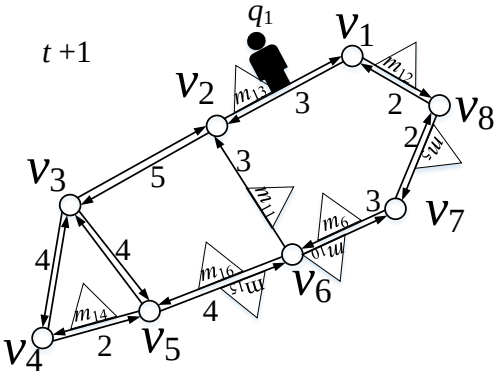


Figure 7. Example of *k*NN Flow Graph Index.

The location information of each moving object m is updated regularly. We use a moving object table to store this location information at a given moment. Figure 8 shows the location information of the moving objects on a road network at times t and $t+1$, where v_{start} and v_{end} are the respective start and end vertices of edge (v_{start}, v_{end}); M is the set of moving objects on this edge; offset contains the offsets of the moving objects to the end vertex v_{end}, corresponding one-to-one with the moving objects in M; and M_N is the number of moving objects in M.



Moving Object List				
v_{end}	M	v_{start}	*Offset*	M_N
v_1	m_{12}	v_2	1	1
v_2	m_{11},m_{13}	v_6,v_3	1,2	2
v_3	m_{14}	v_2	2	1
v_5	m_{16}	v_6	1	1
v_6	m_{10},m_{15}	v_7,v_5	1,1	2
v_7	m_5,m_9	v_8,v_6	1,1.5	2
v_8	m_8,m_6	v_1,v_7	0.5,1	2
...	...	...	...	...



Moving Object List				
v_{end}	M	v_{start}	*Offset*	M_N
v_1	m_{13}	v_2	2	1
v_2	m_{11}	v_6	1	1
v_4	m_{14}	v_5	1	1
v_6	m_6,m_{15}	v_7,v_5	2,1	2
v_7	m_5,m_{10}	v_8,v_6	1,2	2
v_8	m_{12}	v_1	1	1
...	...	...	...	...

Figure 8. Example of moving object list.

Taking Figure 8 as an example, given query point q_1 and assuming that all moving objects are candidate objects, when we initiate a *k*NN query at time t, we first map the query point q_1 to the starting vertex with an offset; i.e., $q_1 \leqslant \langle (v_2, v_1), 1, t \rangle$. Subsequently, all moving objects are offset to the vertices in their end direction, which become the candidate vertices. For example, moving object m_{13} can be represented as $m_{13} \geqslant \langle (v_3, v_2), 2, t \rangle$; i.e., m_{13} is on the edge (v_3, v_2), with an offset of 2 from the vertex v_2. Similarly, each moving object is mapped to a vertex on the road network with an offset, and the final result is as shown in the Moving Object List at time t. In this way, we can use the AWTDN algorithm for *k*NN

queries. For the kNN set, $R = \{v_1, v_2, \cdots, v_K\}$ returned by AWTDN is not the true result. We must continue restoring the vertices in R to moving objects on the road network. Taking v_1 as an example, $M_N = 1$; i.e., there is only one moving object on v_1, i.e., m_{13}. The final distance from m_{13} to the query point q_1 is dist(m_{13}, q_1) = dist(v_2, v_1) + q_1.offset + m_{13}.offset $= 0 + 1 + 2 = 3$. The actual kNN result set is obtained after restoring all vertices to moving objects in turn. It is worth noting that some vertices have more than one moving object, such as vertex v_2, which has moving objects m_{11} and m_{13}. Therefore, it is necessary to further screen the moving objects to obtain the final k results.

At time $t + 1$, as shown, due to the changes in positions of moving objects, the vertices to which moving objects are offset may also change. For example, due to the change in the position of m_{13}, its offset vertex changes from v_2 at time t to v_1 at time $t + 1$. At this point, we need to set $v_{2.isCandidate}$ to false and$v_{1.isCandidate}$ to true. We take Figure 9 as an example to explain the kNN query in a dynamic environment.

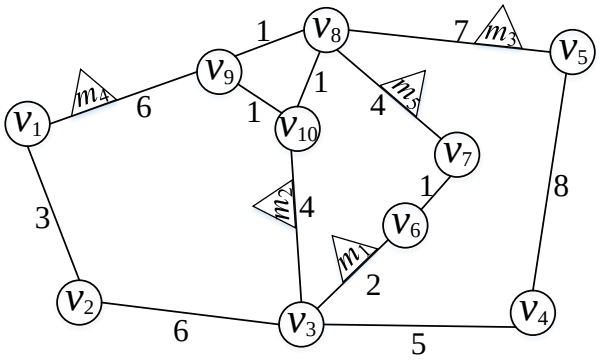


Figure 9. kNN query in dynamic environment.

The starting vertex of query point q in Figure 9 is v_3. At time t, there are five moving objects, m_1, m_2, m_3, m_4, m_5. m_1 is on the edge (v_3, v_6), and the distance between v_3 and v_6 is 2. We map each to a terminating vertex with the offset. Table 3 shows the moving object list at time t after mapping. $Offset$ = 1 means the distance from m_1 to v_6 is 1, M_N = 1 means the number of the moving object on the edge (v_3, v_6) is 1. Then, for the vertices after mapping, we use AWTDN to obtain the result set R and restore its vertices to real moving objects according to the moving object list, so as to obtain the final kNN result. The solution process is shown in Table 4. In the dynamic query, we must modify the kNN result set obtained by AWTDN. Suppose that the 4NN of query point q is 4NN = $\{v_6, v_7, v_{10}, v_4\}$. If a vertex v_8 satisfies dist(v_8,q) = dist(v_4,q), then we continue to save v_8; i.e., the 4NN of q is 4NN = $\{v_6, v_7, v_{10}, v_4, v_8\}$. This ensures that the real moving objects are not missed.

Table 3. List of moving objects at time t.

v_{end}	M	v_{start}	$Offset$	M_N
v_6	m_1	v_3	1	1
v_{10}	m_2	v_3	2	1
v_5	m_3	v_8	3	1
v_1	m_4	v_9	2	1
v_8	m_5	v_7	2	1

A moving object table stores the moving objects' location information at specific time points, including information such as the starting and ending vertices of the edge on which the object is located, the set of moving objects on that edge, the offset of each object from the ending vertex, and the number of moving objects on the edge. This allows the transformation of the problem from dynamic to static, allowing the use of conventional static kNN algorithms.

Table 4. *k*NN solution process of q.

Object	Mapped Vertex	Currently Processed Vertex	...	Min-Heap	Threshold
$\emptyset$	v_3	$(v_3, \emptyset, 0)$		$(v_6, v_3, 2)$	$+\infty$
m_1	v_6	$(v_6, v_3, 2)$	...	$(v_7, v_3, 3)$	$+\infty$
m_1	v_7	$(v_7, v_3, 3)$	...	$(v_{10}, v_3, 4)$	$+\infty$
m_1,m_2	v_{10}	$(v_{10}, v_3, 4)$	...	$(v_4, v_3, 5)$	$+\infty$
m_1,m_2	v_4	$(v_4, v_3, 5)$	...	$(v_8, v_3, 5)$,$(v_1, v_4, 14)$	$+\infty$
m_1,m_2,m_5	v_8	$(v_8, v_3, 5)$	...	$(v_9, v_3, 5)$,$(v_1, v_8, 14)$, $(v_5, v_8, 17)$	$+\infty$
m_1,m_2,m_5	v_9	$(v_9, v_3, 5)$	...	$(v_2, v_3, 6)$,$(v_1, v_9, 14)$, $(v_5, v_9, 17)$	5

5. Experimental Analysis

5.1. Experimental Parameters

We implemented the AWTDN algorithm in the Java language and compared it with TEN-Query [2] and G*Tree [7]. The experimental environment comprised an AMD Ryzen 9 5900X 12-Core CPU at 3.69 GHz, 64 GB of memory, a 4 TB hard drive, and Windows 10.

Dataset: The dataset came from real-world road networks provided by Urban Road Network [24]. Six sets were selected for experiments, as shown in Table 5, where h and w are the respective height and width of average weight tree decomposition.

Table 5. Statistics of road network data.

Dataset	|V|	|E|	h	w
Quanzhou (QZ)	5672	7521	210	79
Dalian (DL)	13,605	17,984	209	65
Pune (PN)	28,649	36,925	359	144
Baghdad (BD)	60,108	88,876	560	201
Tehran (TR)	110,580	147,339	721	356
Bangkok (BK)	154,352	187,798	806	357

Parameter Configuration: For each dataset, we randomly generated candidate objects and controlled their numbers via object density θ, defined as the ratio of the numbers of candidate objects and road network vertices. The configurations of θ- and k-values can be found in Table 6, with default values written in boldface.

Table 6. Parameter Settings.

Parameters	Description	Value
k	Quantity of Nearest Neighbors	**10**, 20, 50, 100
θ	Object Density	0.2, **0.3**, 0.5, 0.7

5.2. Impact of Dataset Size on Algorithm Query Time

We compared the query times of AWTDN, G*Query, and TEN-Query for different dataset sizes, where the object density θ and k take default values. We randomly generated 1000 vertices as query points, using the mean of all query times as the ultimate query time. The results are shown in Figure 10.

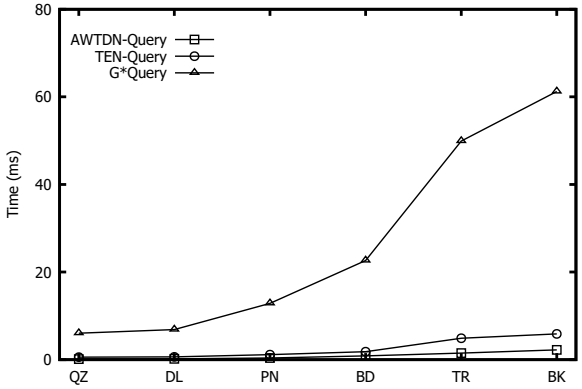


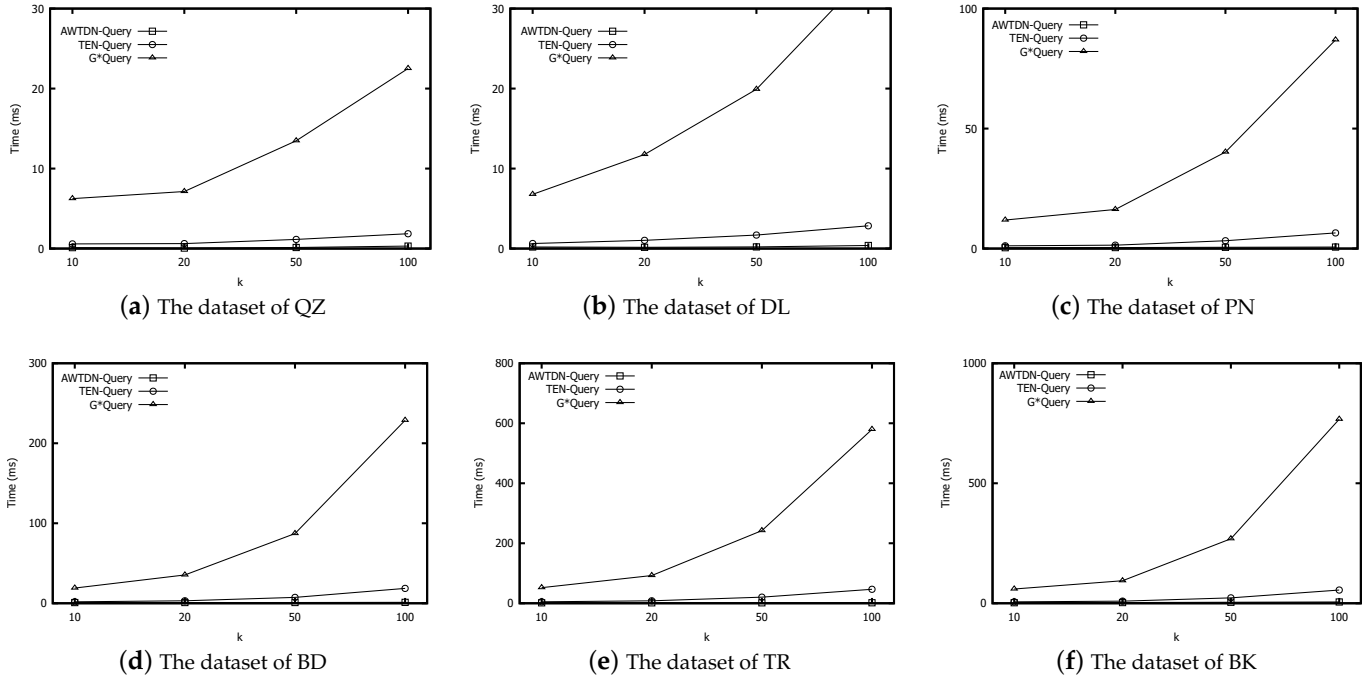
Figure 10. Tree decomposition T_G based on average weight.

It is evident that both AWTDN and TEN-Query are significantly more efficient than G*Query, and AWTDN is the fastest. This is attributed to the avoidance of ineffective computations by applying thresholds. We compared the performance of the algorithms under varying *k*-values.

5.3. Impact of Changing k-Values on Algorithm Query Time

We compared the query efficiencies of the algorithms across six datasets by altering *k*-values, as shown in Figure 11.

The query performance varying different *k*-values is evaluated in Figure 11. A more stationary linearity of our algorithm is measured than G*Query and TEN-Query algorithms as the incremental changes of *k*-values. The flow graph index abstracts all nodes of the graph as the tree nodes, and the size of the tree nodes depends on the quantity of neighbors. Therefore, this tree is not related to the selection of *k*-value, which transcend the *k*-value limitation by storing all potential nodes.



(**a**) The dataset of QZ (**b**) The dataset of DL (**c**) The dataset of PN

(**d**) The dataset of BD (**e**) The dataset of TR (**f**) The dataset of BK

Figure 11. Impact of Changing k-values on Algorithm Query Time.

5.4. Impact of Changing Object Density θ on Algorithm Query Time

We compared the query efficiencies of the three algorithms across six datasets by adjusting θ values, with results as shown in Figure 12.

It is apparent that, irrespective of changes in the θ value, the performance of AWTDN across all six datasets is markedly superior to that of TEN-Query and G*Query, thereby confirming the universal applicability of AWTDN.

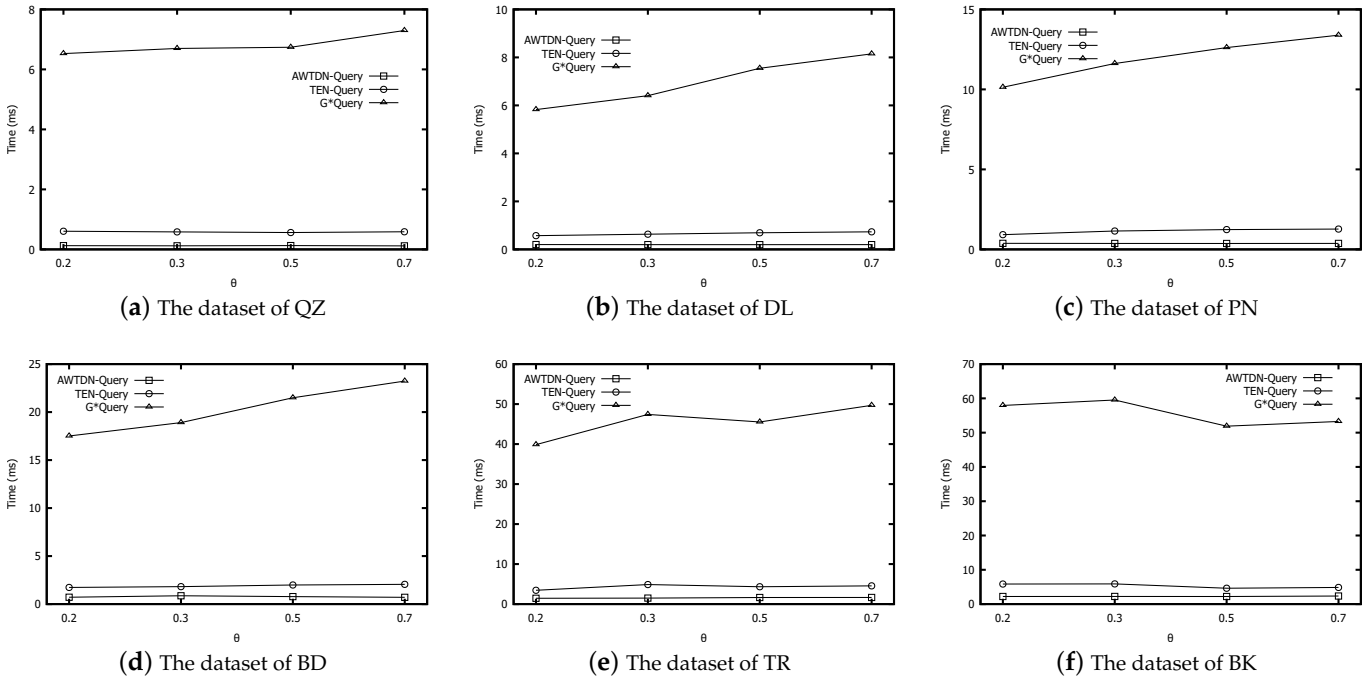


Figure 12. Impact of Changing Object Density θ on Algorithm Query Time.

5.5. Comparison of Accessed Vertex Numbers across Different Datasets

We compared the number of vertices accessed by AWTDN, TEN-Query, and G*Query across six datasets. For each dataset, we generated 1000 random query points and took the average total number of accessed vertices as the final count. Both k and θ took default values. The comparison results are shown in Figure 13.

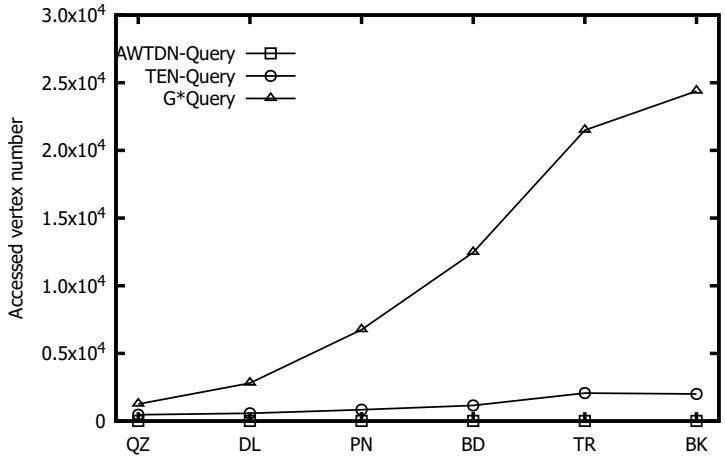


Figure 13. Comparison of the number of visited vertices.

The results reveal that AWTDN accesses significantly fewer vertices than TEN-Query and G*Query, which is attributable to the efficient pruning capacity of the threshold.

5.6. Comparison of Index Construction Time and Space across Different Datasets

We compared the index construction time and space of AWTDAC, TEN-Index, and G*Query across datasets of varying sizes, where the object density θ takes the default value of 0.3. Since the construction of TEN-Index depends on k, to accommodate queries of all k-values, we set k to 100. It is worth mentioning that AWTDAC does not rely on k and thus can satisfy kNN queries of any k-value. We set the parameters of G*Tree according to the original text. The comparison results are depicted in Figures 14 and 15.

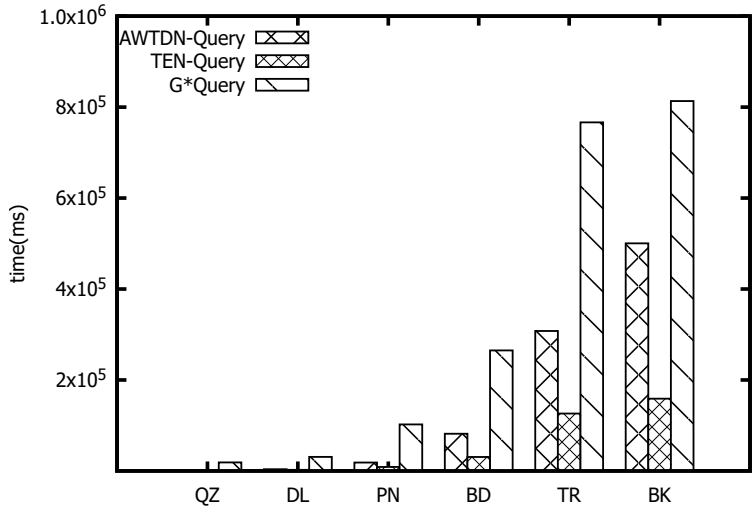


Figure 14. Comparison of index construction time.

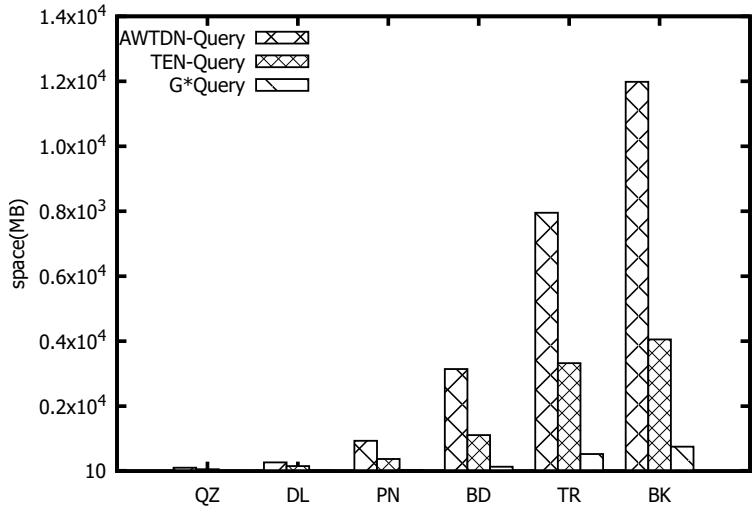


Figure 15. Comparison of index construction space.

AWTDAC incurs higher construction time and space costs than TEN-Index, which only stores the kNN results within the contracted neighbor range, which explains why TEN-Index depends on the k-value. In contrast, AWTDAC stores all vertices within the subtree scope, thus breaking the k-value limitation and enabling kNN queries of any k-value. Although G*Tree has the smallest space cost, it has the largest time cost. G*Tree employs the Dijkstra algorithm to calculate the shortest path network distance, while AWTDAC and TEN-Index use H2H-Query to measure the road network distance, which is significantly more efficient than Dijkstra.

6. Conclusions

*k*NN queries for road networks are studied in this paper. Firstly, a flow graph index is proposed to store the intermediated results, which is generated from a graph-decomposed tree. Rules obtained from the flow graph index are found to reduce the quantity of candidate nodes. Secondly, the *k*NN query algorithm is used to conduct the final results on this flow graph index. Finally, the effectiveness and efficiency of our proposed algorithm are confirmed through experiments.

Author Contributions: Conceptualization, W.J.; methodology, M.B.; software, F.W.; validation, X.W.; formal analysis, G.L.; data curation, B.N.; writing—original draft preparation, F.W.; writing—review and editing, W.J.; supervision, G.L.; funding acquisition, G.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China under Grant (No. 61976032 and No. 62002039), the genReral scientific research project of Liaoning (No. LJKZ0063).

Data Availability Statement: The data used to support this study are available from the corresponding author upon request.

Acknowledgments: I would like to extend my sincere gratitude to Yangjie Zhou from Rizhao Institute of Metrology for his invaluable assistance throughout the process of writing this paper. Zhou's expertise and advice played a significant role in enhancing the quality of the paper. I am deeply appreciative of the time and effort generously provided by Zhou, which greatly facilitated the completion of this paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dijkstra, E.W. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*; ACM Digital Library: New York, NY, USA, 2022; pp. 287–290.
2. Ouyang, D.; Wen, D.; Qin, L.; Chang, L.; Zhang, Y.; Lin, X. Progressive top-*k* nearest neighbors search in large road networks. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 14–19 June 2020; pp. 1781–1795.
3. He, D.; Wang, S.; Zhou, X.; Cheng, R. An efficient framework for correctness-aware *k*NN queries on road networks. In Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE), Macao, China, 8–11 April 2019; pp. 1298–1309.
4. Tianyang, D.; Lulu, Y.; Qiang, C.; Bin, C.; Jing, F. Direction-aware *K*NN queries for moving objects in a road network. *World Wide Web* **2019**, *22*, 1765–1797. [CrossRef]
5. Shen, B.; Zhao, Y.; Li, G.; Zheng, W.; Qin, Y.; Yuan, B.; Rao, Y. V-tree: Efficient *k*nn search on moving objects with road-network constraints. In Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering (ICDE), San Diego, CA, USA, 19–22 April 2017; pp. 609–620.
6. Zhong, R.; Li, G.; Tan, K.L.; Zhou, L.; Gong, Z. G-tree: An efficient and scalable index for spatial search on road networks. *IEEE Trans. Knowl. Data Eng.* **2015**, *27*, 2175–2189. [CrossRef]
7. Li, Z.; Chen, L.; Wang, Y. G-tree: An efficient spatial index on road networks. In Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE), Macao, China, 8–11 April 2019; pp. 268–279.
8. Dan, T.; Luo, C.; Li, Y.; Guan, Z.; Meng, X. LG-Tree: An Efficient Labeled Index for Shortest Distance Search on Massive Road Networks. *IEEE Trans. Intell. Transp. Syst.* **2022**, *22*, 23721–23735. [CrossRef]
9. Bao, J.L.; Wang, B.; Yang, X.C.; Zhu, H.J. Nearest neighbor query in road networks. *J. Softw.* **2018**, *29*, 642–662.
10. Li, L.; Cheema, M.A.; Ali, M.E. Continuously monitoring alternative shortest paths on road networks. In *VLDB Endowment*, 3rd ed.; Association for Computing Machinery: New York, NY, USA, 2020; pp. 2243–2255.
11. Huang, Y.-K.; Lee, C.-P.; Tsai, C.-Y. Evaluating *K*NN-skyline queries in dynamic road networks. In Proceedings of the 2018 27th Wireless and Optical Communication Conference (WOCC), Hualien, Taiwan, 30 April–1 May 2018; pp. 1–2.
12. Halin, R. S-functions for graphs. *J. Geom.* **1976**, *8*, 171–186. [CrossRef]
13. Ouyang, D.; Wen, D.; Qin, L.; Chang, L.; Lin, X.; Zhang, Y. When hierarchy meets 2-hop-labeling: Efficient shortest distance and path queries on road networks. In Proceedings of the SIGMOD'18: Proceedings of the 2018 International Conference on Management of Data, Paris, France, 13–18 June 2004; pp. 1–25.
14. Cho, H.J.; Chung, C.W. An efficient and scalable approach to CNN queries in a road network. In Proceedings of the International Conference on VLDB, Trondheim, Norway, 30 August–2 September 2005; pp. 865–876.
15. Safar, M. Enhanced Continuous KNN Queries Using PINE on Road Networks. In Proceedings of the 2006 1st International Conference on Digital Information Management, Bangalore, India, 6 December 2007; pp. 248–256.

16. Fu, Q.; Sun, G.; Zhang, Z. An efficient precomputation technique for approximation distance query in road networks. In Proceedings of the 2013 IEEE 14th International Conference on Mobile Data Management, Milan, Italy, 3–6 June 2013; pp. 131–135.
17. Abeywickrama, T.; Cheema, M.A.; Storandt, S. Hierarchical Graph Traversal for Aggregate k Nearest Neighbors Search in Road Networks. In Proceedings of the International Conference on Automated Planning and Scheduling, Nancy, France, 14–19 June 2020; pp. 2–10.
18. Bareche, I.; Xia, Y. A Distributed Hybrid Indexing for Continuous KNN Query Processing over Moving Objects. *ISPRS Int. J. Geo-Inf.* **2022**, *11*, 264. [CrossRef]
19. Chen, Y.J.; Chuang, K.T.; Chen, M.S. Coupling or decoupling for KNN search on road networks? a hybrid framework on user query patterns. In Proceedings of the 20th ACM International Conference on Information and Knowledge Management, New York, NY, USA, 24–28 October 2011; pp. 795–804.
20. Hlaing, A.T.; Htoo, H.; Ohsawa, Y. Efficient Reverse *k*NN Query Algorithm on Road Network Distances Using Partitioned Subgraphs. In *Advances in Conceptual Modeling: ER 2014 Workshops, ENMO, MoBiD, MReBA, QMMQ, SeCoGIS, WISM, and ER Demos, Atlanta, GA, USA, 27–29 October 2014*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 212–217.
21. Luo, S.; Kao, B.; Li, G.; Hu, J.; Cheng, R.; Zheng, Y. Toain: A throughput optimizing adaptive index for answering dynamic *k*NN queries on road networks. *Proc. Vldb Endow.* **2018**, *11*, 594–606. [CrossRef]
22. Li, C.; Gu, Y.; Qi, J.; He, J.; Deng, Q.; Yu, G. A GPU Accelerated Update Efficient Index for *k*NN Queries in Road Networks. In Proceedings of the 2018 IEEE 34th International Conference on Data Engineering (ICDE), Paris, France, 16–19 April 2018; pp. 881–892.
23. Sun, Y.; Li, G.; Du, J.; Ning, B.; Chen, H. A subgraph matching algorithm based on subgraph index for knowledge graph. *Front. Comput. Sci.* **2020**, *16*, 163606. [CrossRef]
24. Urban Road Network Data. Figshare. Dataset. Available online: https://figshare.com/articles/dataset/Urban_Road_Network_Data/2061897/1 (accessed on 30 September 2023).