Efficient kNN Search in Public Transportation Networks

Qingshuai Feng
University of New South Wales
qsfeng@cse.unsw.edu.au

Junhua Zhang*
University of New South Wales
junhua.zhang@unsw.edu.au

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

Lu Qin
University of Technology Sydney
lu.qin@uts.edu.au

Ying Zhang
University of Technology Sydney
ying.zhang@uts.edu.au

Xuemin Lin
Shanghai Jiao Tong University
xuemin.lin@gmail.com

ABSTRACT

Public transportation plays a vital role in mitigating traffic congestion and reducing carbon emissions. The *Top-k Nearest Neighbor* (kNN) search in public transportation networks is a fundamental problem in location-based services, which aims to find k nearest objects from a given query point. The traditional method, Dijkstra's algorithm has been employed to tackle the kNN problem, however, it is notably inefficient in processing queries. While other works precompute an index to speed up query processing. However, they are still slow in processing queries. Furthermore, they cannot scale to large graphs due to their reliance on resource-intensive path indexes. To address these limitations, we introduce a novel index-based approach that utilizes a simple yet effective index structure to handle kNN queries with a near-optimal time complexity. The index does not rely on a path index, making it efficient to construct and scalable to large graphs. Extensive experiments are conducted on real-world datasets to demonstrate the efficiency and scalability of our approach. The results show that our approach outperforms existing solutions by up to four orders of magnitude in query processing and two orders of magnitude in index construction.

PVLDB Reference Format:
Qingshuai Feng, Junhua Zhang, Wenjie Zhang, Lu Qin, Ying Zhang, and Xuemin Lin. Efficient kNN Search in Public Transportation Networks. PVLDB, 17(11): 3402 - 3414, 2024.
doi:10.14778/3681954.3682009

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://rb.gy/98eziv.

1 INTRODUCTION

Public transportation plays a vital role in modern cities, providing a convenient and efficient way of transportation while reducing traffic congestion and pollution [27, 29]. The public transportation network (PTN) consists of a large number of stations and vehicles operating on a fixed schedule. It can be modeled as a directed multi-graph $G = (V, E)$, where V is the set of all stations and E is the set of individual trips with fixed departure and arrival times between adjacent stations. The *Top-k Nearest Neighbor (kNN)* search in public transport networks is a fundamental problem in location-based services. Let $\mathcal{P} \subset V$ be a set of objects in the network G, given a query point (vertex) q, a departure time t and a positive integer k, the kNN query aims to find the k objects in $\mathcal{P}$ that are closest to q in terms of their arrival times.

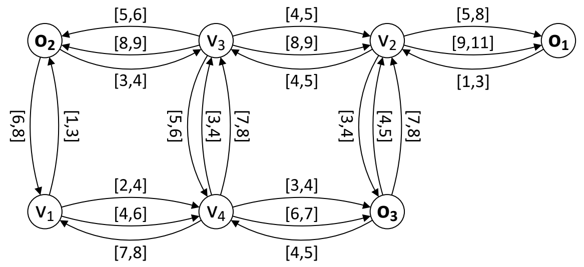


Figure 1: An example of a public transportation network G

kNN search has numerous applications across different domains. In property rental services such as *Domain* and *Beike*, a tenant usually prefers houses with shorter public transportation commute times to work or school. These platforms can use the house information and the commute times to offer recommendations for users according to their office or school locations, making their house-hunting process more efficient. In travel guidance service platforms like *TripAdvisor* and *Dianping*, kNN search in PTNs can significantly enhance user experience. When tourists seek nearby attractions accessible by public transportation, the platforms can provide recommendations of these attractions based on the location of the tourists by conducting a kNN search of these attractions. In the above examples, the objects, like the houses and attractions, are typically fixed points of interest in their platforms.

Existing Solutions. Numerous solutions have been proposed to solve the kNN problem in transportation networks [7, 8, 13, 15–18, 21, 23, 24, 26, 33, 36]. While most of them are designed for road networks, only a few of them are specifically tailored for public transportation networks. Li et al. [18] employs Dijkstra's algorithm [9] to find k objects that are closest to the query point. This method is inefficient since it might traverse a large number of vertices and edges, especially when the graph is large and objects are far from the query point. Several works [10, 14, 17] resort to an index to speed up the kNN query. We review them as follows.

Huang et al. [14] proposed the TFS for the top-k nearest keyword search problem in public transportation networks. It can be used for the kNN query by limiting the number of keywords to 1. TFS is built on a path index [28] designed for the *Earliest Arrival Path* queries. When answering a query, TFS computes the earliest arrival time

*Junhua Zhang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682009

of each object $o \in \mathcal{P}$ using TTL and returns the k objects with the earliest arrival times. Efentakis et al. [10] proposed PTLDB for the kNN query in PTN. PTLDB is also built on a path index [28]. Unlike the TFS algorithm that computes the earliest arrival time of each object in $\mathcal{P}$, PTLDB creates an auxiliary index for each hub vertex h to store the kNN of h. When answering a kNN query, PTLDB uses the kNN of each hub h of the query vertex u to find the kNN of u, where the earliest arrival paths from u to h are stored in the TTL index. Li et al. [17] proposed TD-GLAD for the kNN queries in time-dependent road networks, which can be applied to PTNs. Like TFS, TD-GLAD also contains a path index for the earliest arrival time query, but it partitions the graph into grids according to the geographical information and stores an objects list for each grid. During the kNN query, the path queries are only conducted for the objects in the grid, the search space is limited to the grids that are close to the query vertex.

Limitations of Existing Solutions. The limitations of the existing solutions are summarized as follows. These limitations exist not only in the works for public transportation networks but also in the works for road networks. However, they are not well addressed.

Query efficiency. The query efficiency of existing solutions is not satisfactory. Dijkstra's algorithm based methods or traversal-based methods with heuristics can handle queries in large graphs, but they are notably slow in processing queries. The index-based methods, for example, TFS and PTLDB for PTNs, TD-GLAD [17] for time-dependent networks, GLAD [13], TOAIN [21] and TEN-index [23] for road networks, are also inefficient in processing queries. These index-based methods either rely on the path indexes for distance/earliest arrival time computation or build partial kNN indexes on path indexes for faster query processing. They still require considerable time to answer queries.

Scalability. As analyzed above, almost all existing methods rely on the path index for fast query processing. However, the path index is resource-intensive and requires a large amount of time to construct and space to store. As a result, they cannot scale to large graphs. For example, both TFS and PTLDB failed to construct the index for large PTNs in Buenos Aires and the UK in our experiments (see Section 8). This is because their underlying TTL index requires too much time to construct. The index-based methods for road networks also suffer from the same problem.

Challenges and Our Solution. In light of the limitations of existing works, we make the following observations: the kNN query is different from the path queries, the kNN query is centered at the query vertex q, while the path query is between two vertices u and v; the number of possible path queries is quadratic to the number of vertices in G, while the number of possible kNN queries is linear to the number of vertices in G. From this perspective, building an index for the kNN problem based on a resource-intensive path index is too cumbersome.

Based on the above observations, we propose a novel index, named TNN-Index, for efficient kNN query in public transportation networks. The main idea of TNN-Index is simple: for each vertex v in G, it stores the kNN of v for some critical departure times t_d. When processing a query $Q(v, t, k)$, TNN-Index finds the kNN of v with the minimum critical departure time t_d such that $t \leq t_d$, then the corresponding result is directly returned. Although the index seems simple, how to construct it efficiently is non-trivial:

- One solution is to employ the Tree Decomposition technique as it has been widely used for constructing indexes in graphs [5, 17, 18, 22, 23, 31, 34, 35] for path and kNN queries. However, the existing methods that use tree decomposition for kNN queries contain heavy path indexes and constructing a path index using tree decomposition is time-consuming. Hence, how to employ tree decomposition to build a kNN index way more efficiently than building a path index remains an open problem.
- In PTN, there are hundreds of multi-edges between two vertices, how to handle them efficiently is challenging, as is also noted in the last paragraph of Section 6.2.1 in literature [17]: the index size and index construction time may increase significantly as the number of multi-edges increases.

To address these challenges, we propose a novel and efficient index construction framework based on the planarity property of PTN and implement the framework with efficient algorithms based on the notion of Tree Decomposition.

Contributions. We summarize our contributions as follows:

- *A new index for efficient kNN queries in large public transportation networks.* We propose a simple but effective index, TNN-Index, with which, the query processing achieves exceptional speed. Compared with the existing solutions, our algorithm is up to four orders of magnitude faster in query processing time.
- *An efficient index construction algorithm.* We design a novel index construction framework to exploit the planarity property of PTN and implement it efficiently using Tree Decomposition. The empirical evaluation shows that the index construction time outperforms existing methods by up to two orders of magnitude and scales well in large graphs.
- *Extension of* TNN-Index *and index compression.* To incorporate the scenarios where objects have specified opening hours, we extend TNN-Index by exploring a new notion of *Earliest Accessible Time.* Additionally, we introduce an index compression technique to reduce the index size while preserving query efficiency.
- *Extensive empirical studies.* We conduct comprehensive experiments on 8 real-world PTNs and 2 synthetic networks which verify the efficiency and scalability of our approach.

Organization. Section 2 presents the preliminary and Section 3 reviews existing solutions. Section 4 introduces our index structure and query processing algorithm. The index construction algorithm is elaborated in Section 5. Section 6 extends the index. Section 7 presents an index compression technique. Section 8 evaluates the proposed algorithms. Related works are discussed in Section 9. Section 10 concludes the paper.

2 PRELIMINARY

A Public Transportation Network (PTN) can be modeled as a directed multi-graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. Each vertex $v \in V$ represents a station, and each edge $e = \langle u, v, t_d, t_a \rangle \in E$ represents a vehicle departing from station u at time t_d and arriving at station v at time t_a. Multiple edges may exist between two vertices, which corresponds to multiple trips with different departure and arrival times. We say that $e = \langle u, v, t_d, t_a \rangle$

Table 1: Notations

Notation	Description
$E_{out}(u)$ / $E_{in}(u)$	set of all edges that start from u / end at u
$d^+(u)$ / $d^-(u)$	out-degree / in-degree of u
$V_k(u,t)$	kNN of u departing no sooner than t
$T_d(u)$	set of departure times of u
$\mathcal{K}(u)$	TNN-Index of u
T_G	tree decomposition of G
$X(v)$	tree node of v in T_G
$\mathsf{E_p}(u,v)$	edge profile from u to v

is an *outgoing* edge of u and an *incoming* edge of v, accordingly, v is an *out-neighbor* of u and u is an *in-neighbor* of v. We use $E_{out}(v)$ (resp. $E_{in}(v)$) to denote the set of all edges that start from v (resp. end at v). Similarly, we use $N_{out}(v)$ and $N_{in}(v)$ to denote the set of all out-neighbors and in-neighbors of v, respectively. If the subscripts of these notations are omitted, they represent the union of the notations with subscripts, i.e., $E(v) = E_{out}(v) \cup E_{in}(v)$ and $N(v) = N_{out}(v) \cup N_{in}(v)$. The out-degree and in-degree of v are denoted as $d^+(v) = |E_{out}(v)|$ and $d^-(v) = |E_{in}(v)|$, respectively. We use $\overline{d^+}$ and $\overline{d^-}$ to denote the average out-degree and in-degree of the graph, respectively. Notations are summarized in Table 1.

*Definition 2.1 (**Path**).* A path p in G is a sequence of edges $(e_1, e_2, \cdots, e_k)$, such that for any $i \in [1, k-1]$, the ending vertex of e_i equals the starting vertex of e_{i+1}, and $t_a(e_i) \le t_d(e_{i+1})$. The departure (resp. arrival) time of p is defined as $t_d(p) = t_d(e_1)$ (resp. $t_a(p) = t_a(e_k)$).

*Definition 2.2 (**Earliest Arrival Time/Path**).* Given two vertices u and v in G and a starting time t, the *Earliest Arrival Time* (*EAT*), denoted as $EAT(u,v,t)$, is defined as the earliest time one can arrive at v from u departing no sooner than t, the corresponding path is called the *Earliest Arrival Path*. If there is no path from u to v departing no sooner than t, then $EAT(u,v,t) = \infty$.

We consider a set of objects $\mathcal{P}$ in the G, representing the points of interest (POI), which function as the goal of the kNN search. To simplify the discussion, we assume that each object $o \in \mathcal{P}$ is located at a vertex $v \in V$. For objects that are not located at a vertex, we can add a vertex at the location of the object and add edges between the objects and other vertices that are reachable to the object (e.g., the stations within walking distance of the object). The edge is labeled by the time t (e.g., walking time) required to reach the object, denoting that anyone starting from the vertex at time t_d can reach the object at time $t_d + t$. For the sake of simplicity, we also assume that there is at most one object located at a vertex. If there are multiple objects at the same vertex v, we can view them as different vertices with 0 walking distance from v.

Problem Statement. In this paper, we abuse the term kNN to refer to the *k-Earliest Arrival Neighbors* and aim to answer the kNN queries efficiently.

*Definition 2.3 (**kNN Query**).* Given a graph $G = (V, E)$ containing a set of objects $\mathcal{P} \subset V$, a kNN query $Q(u,t,k)$ specifies a query vertex u, a starting time t, an integer k, and aims to compute a set, denoted as $V_k(u,t)$, containing k vertices and their corresponding earliest arrival times, such that:

(1) $|V_k(u,t)| = k$;
(2) for any $v \in V_k(u,t) \Rightarrow v \in \mathcal{P}$;
(3) $\forall v \in V_k(u,t), v' \in \mathcal{P} \backslash V_k(u,t)$, $EAT(u,v,t) \le EAT(u,v',t)$.

Example 2.4. Figure 1 is an example of a public transportation network with 7 vertices and 23 edges. Among these vertices, o_1, o_2 and o_3 are objects. The edge $\langle v_4, v_1, 7, 8\rangle$ indicates that a vehicle departs from v_4 at time 7 and arrives at v_1 at time 8. $EAT(v_1, v_2, 3) = 8$ since the earliest arrival time from v_1 to v_2 departing no sooner than time 3 is 8, which is achieved by the path $(\langle v_1, v_4, 4, 6\rangle, \langle v_4, o_3, 6, 7\rangle, \langle o_3, v_2, 7, 8\rangle)$. Given a query vertex v_1, a starting time $t = 1$, the kNN query with $k = 2$ returns $V_k(v_1, 1) = \{(o_2, 3), (o_3, 7)\}$.

For ease of presentation, we assume that the outgoing (resp. incoming) edges of the same vertex have different departure (resp. arrival) times. With this assumption, the earliest arrival time is associated with a unique path. The result of any kNN query is also unique. In scenarios where timestamps are not inherently unique, a practical workaround is to introduce a small random increment (resp. decrement) to each departure time (resp. arrival time). This minor adjustment guarantees uniqueness without compromising the integrity of the results.

3 EXISTING SOLUTIONS

3.1 Index-Free Approach

A straightforward method for processing the kNN query is to use an online search algorithm to traverse the graph. A Dijkstra-based algorithm INE [24] has been proposed for the kNN query in road networks and [18] adapted this algorithm to address the kNN query in public transportation networks. Given a query vertex u and a starting time t_d, the INE visits vertices in order of their earliest arrival time from u and terminates once k objects are collected.

Limitations of INE. The INE algorithm needs to traverse the graph for kNN query processing. If there are many objects in the graph, then the INE algorithm can find the kNN efficiently as the algorithm can terminate by visiting only a small number of vertices. However, if the number of objects is small, then the objects are likely to be distant from the query vertex u, and the INE algorithm needs to traverse a large portion of the graph to find the kNN.

3.2 Index-Based Approaches

To improve the kNN query efficiency, several index-based approaches have been proposed [10, 14, 17]. As all of these approaches are based on the 2-hop indexes [22, 28] for path queries, we first introduce the 2-hop index.

2-hop index [6] has been widely used for path queries in graphs [4, 11, 12, 19, 20, 22, 28] and was extended for PTNs and time-dependent networks by Wang et al. [28] and Li et al. [17], respectively. Given a graph G, the 2-hop path index assigns to each vertex $v \in V$ an out-label $L_{out}(v)$ and an in-label $L_{in}(v)$. Each entry (w, t_d, t_a) in $L_{out}(v)$ (resp. $L_{in}(v)$) indicates an earliest arrival path p from v to w (resp. from w to v) departing at time t_d and arriving at time t_a, w is called a *hub* of v. The hub w is selected for covering as many paths as possible to reduce index size. To achieve this goal, one of the commonly adopted heuristics is to rank the vertices based on their degrees and select the vertices with higher ranks (larger degrees) as hubs, a path entry (w, t_d, t_a) is added to $L_{out}(v)$ if and only if it is an earliest arrival path from v to w and w has the highest rank among all the vertices on the path.

Given an *EAT*query $EAT(u,v,t)$, the TTL employs $L_{out}(u)$ and $L_{in}(v)$ to compute the earliest arrival time of v departing at t with the following two steps:

1. if v is in $L_{out}(u)$ or u is in $L_{in}(v)$, then find entires (v, t_d, t_a) in $L_{out}(u)$ or entries (u, t_d, t_a) in $L_{in}(v)$ such that $t \leq t_d$ and t_a is minimized, and
2. enumerate each entry (w, t_d, t_a) in $L_{out}(u)$ and each entry (w, t'_d, t'_a) in $L_{in}(v)$ of their common hubs w such that $t \leq t_d$, $t_a \leq t'_d$ and t'_a is minimized.

If both steps find a feasible path, then the earliest arrival time is the minimum of the two arrival times. If none of the steps find a feasible path, then the earliest arrival time is ∞.

TFS. Based on TTL, Huang et al.[14] developed the Temporal Forward Search (TFS) algorithm. TFS is designed for top-k Nearest keyword Searches in public transportation networks, but it can also be used for the kNN query. Given a query, TFS operates in three phases: (1) acquiring the list of vertices $\mathcal{P}$ where the objects are located, (2) computing the earliest arrival time between the query vertex q and each vertex v in $\mathcal{P}$ using TTL, and (3) sorting and returning the top-k results. Although TTL-Query is capable of rapid responses, the need to conduct $|\mathcal{P}|$ such queries can become a bottleneck in efficiency, especially when the size of $\mathcal{P}$ is large.

PTLDB. Efentakis [10] proposed PTLDB (Public Transportation Labeling on Databases) framework with TTL as the underlying layer. Different from TFS, PTLDB creates an auxiliary inverse label $L^{-1}_{in}(h)$ for each hub h. Specifically, given the TTL index, for each object $o \in \mathcal{P}$ and each entry $(h, t_d, t_a) \in L_{in}(o)$, it adds an entry (o, t_d, t_a) to $L^{-1}_{in}(h)$. Then it groups the entries in $L^{-1}_{in}(h)$ by t_d and sorts the entries in each group by t_a, only the top-k entries are kept in each group. To answer a kNN query $Q(u, t, k)$, PTLDB enumerates each entry (h, t_d, t_a) in $L_{out}(u)$ where $t \leq t_d$ and each entry (o, t'_d, t'_a) in $L^{-1}_{in}(h)$ where $t_a \leq t'_d$, and returns the k objects with the smallest t'_a. This method removes the need to conduct $|\mathcal{P}|$ queries, but it is still inefficient as it is built on the expensive TTL index which requires a large amount of space and time to construct.

TD-GLAD. Li et al. [17] extended He et al.'s work [13] and proposed TD-GLAD to answer kNN queries on time-dependent networks. TD-GLAD can also be used for PTNs as the PTN is a special type of time-dependent networks where the cost function is a disjoint piecewise linear function. Like TFS and PTLDB, TD-GLAD is also based on a 2-hop path index, but it partitions the network into grids using geographical information and stores a list of objects for each grid. When answering a kNN query, it first computes the earliest arrival time of objects in the grid where the query vertex is located and then expands to neighbor grids to find more objects. The search will be terminated if the earliest arrival time of the k-th nearest object currently found is no later than the possible earliest arrival time between the query vertex and the boundary of the explored grids. Although TD-GLAD can reduce the number of earliest arrival time queries and the search space, it has similar drawbacks as TFS and PTLDB in that it is built on the expensive 2-hop path index.

4 A NEW INDEX-BASED APPROACH

As discussed in Section 3, the existing solutions for the kNN problem in public transportation networks are either based on online search, which is slow in processing queries, or building an index in advance to make the query faster, which requires considerable time to construct and a large space for storing the index. To overcome these limitations and fill the research gaps, we present a new index-based approach, termed as TNN-Index (Timetable graph kNN index), with a low preprocessing time and space cost, while also ensuring real-time query response time. In this section, we will introduce the TNN-Index and the query algorithm. The index construction algorithm will be presented in Section 5.

4.1 TNN-Index

To facilitate efficient kNN queries, a straightforward idea is to store the result set $V_k(v, t)$ for each possible kNN query $Q(v, t, k)$ in the index, and return $V_k(v, t)$ directly when $Q(v, t, k)$ is issued. However, this approach is not practical due to the infinite possible departure times in a query $Q(v, t, k)$. Considering the graph is a real public transportation network, we have the following observations:

- A vertex has a finite set of *departure times* $T_d(v) = \{t_d | (v, u, t_d, t_a) \in E\}$, a passenger can only leave at these times. Therefore, considering only the departure times of a vertex is sufficient.
- Consider two queries $Q(v_1, 2, 2)$ and $Q(v_1, 4, 2)$ in Figure 1, their query results are the same: $Q(v_1, 2, 2) = Q(v_1, 4, 2) = \{(o_3, 7), (o_2, 9)\}$. This is because no matter whether we depart o_1 at time 2 or 4, we have to wait at v_4 until time 6 since 6 is the earliest departure time of v_4 after time 4. In this case, although time 2 is a valid departure time for v_1, it is not *critical* for v_1 in the sense that removing it from the index does not affect the kNN query result.

*Definition 4.1 (**TNN-Index**).* Given a graph G, the TNN-Index of G maintains a data structure $\mathcal{K}(v)$ for each vertex $v \in V(G)$, which is defined for two scenarios:

- If $v \notin \mathcal{P}$, $\mathcal{K}(v)$ stores the kNN $V_k(v, t_d)$, referred as $\mathcal{K}(v, t_d)$, of v for some selected departure times $t_d \in T_d(v)$ such that there is no $V_k(v, t'_d) \in \mathcal{K}(v)$ with $t'_d > t_d$ but $V_k(v, t'_d) = V_k(v, t_d)$.
- Otherwise, $\mathcal{K}(v)$ stores $V_k(v, t_d) \setminus \{(v, t_d)\}$, referred as $\mathcal{K}(v, t_d)$.

We don't store (v, t_d) in $\mathcal{K}(v, t_d)$ if $v \in \mathcal{P}$ because whenever the departure time is, the passenger can always reach v immediately. Whereas the index only considers a small set of departure times and cannot store infinite possible departure times t in a query $Q(v, t, k)$.

Example 4.2. Table 2 presents the TNN-Index for graph G in Figure 1 with $k = 3$. For v_4, there are 3 vertex sets in $\mathcal{K}(v_4)$, each associated with its respective departure times $t_d = 3, 6, 7$. But for v_1 with $T_d(v_1) = \{1, 2, 4\}$, the index omits $t_d = 2$ due to $\mathcal{K}(v_1, 2) = \mathcal{K}(v_1, 4)$. The indexes for objects o_1, o_2, and o_3 do not include themselves.

THEOREM 4.3. *The size of* TNN-Index *is bounded by* $O(n \cdot \overline{d^+} \cdot k)$.

PROOF. There are n vertices in total. For each vertex v, the average size of $T_d(v)$ is limited by $\overline{d^+}$, which is the average out-degree. Furthermore, for each of these departure times, we maintain a set of size $O(k)$. Hence, the size of TNN-Index is $O(n \cdot \overline{d^+} \cdot k)$. □

4.2 Query Processing

With the TNN-Index in place, we introduce the query algorithm, named TNN-Query, in Algorithm 1. The query process is straightforward owing to the comprehensive data structures stored for each vertex in the TNN-Index. Specifically, given a query with a vertex u, a departure time t and an integer k, we first check if u is an object vertex, and if it is, we add it to the result with the departure time t (lines 2-3). Subsequently, we identify the smallest t_d such that t_d is

Table 2: TNN-Index **of** G **with** $k = 3$

v	t_d	$\mathcal{K}(v, t_d)$
$\mathcal{K}(v_1)$	1	$(o_2, 3)$, $(o_3, 7)$, $(o_1, 8)$
	4	$(o_3, 7)$, $(o_2, 9)$, $(o_1, 11)$
$\mathcal{K}(v_2)$	3	$(o_3, 4)$, $(o_2, 6)$, $(o_1, 8)$
	4	$(o_2, 6)$, $(o_3, 7)$, $(o_1, 8)$
	5	$(o_1, 8)$
	9	$(o_1, 11)$
$\mathcal{K}(v_3)$	4	$(o_2, 6)$, $(o_3, 7)$, $(o_1, 8)$
	5	$(o_2, 6)$, $(o_3, 7)$, $(o_1, 11)$
	8	$(o_2, 9)$, $(o_1, 11)$
$\mathcal{K}(v_4)$	3	$(o_3, 4)$, $(o_2, 6)$, $(o_1, 8)$
	6	$(o_3, 7)$, $(o_2, 9)$, $(o_1, 11)$
	7	$(o_2, 9)$, $(o_1, 11)$
$\mathcal{K}(o_1)$	1	$(o_3, 4)$, $(o_2, 6)$
$\mathcal{K}(o_2)$	3	$(o_3, 7)$, $(o_1, 8)$
$\mathcal{K}(o_3)$	4	$(o_1, 8)$, $(o_2, 9)$
	7	$(o_1, 11)$

not earlier than t and $\mathcal{K}(u, t_d)$ exists (line 4). If such a t_d is found, we add $V_k(u, t_d)$ to the result (lines 5, 6). Otherwise, it means that there is no departure time t_d that is not earlier than t that can reach other objects from u. Finally, we return the result (line 7).

Algorithm 1: TNN-Query(u, t, k)

```
1 $V_k \leftarrow \emptyset$;
2 if $u \in \mathcal{P}$ then
3     $V_k \leftarrow \{(u, t)\}$;
4 $t_d \leftarrow \min\{t_d | \forall \mathcal{K}(u, t_d)$, where $t_d \geq t\}$;
5 if $t_d \neq \infty$ then
6     $V_k \leftarrow V_k \cup \mathcal{K}(u, t_d)$;
7 return $V_k$;
```

Theorem 4.4. *The time complexity of Algorithm 1 is* $O(\log \overline{d^+} + k)$.

Proof. The time required to search for t_d is $O(\log \overline{d^+})$ by a binary search. Subsequently, retrieving kNN incurs a cost of $O(k)$. Therefore, the time complexity of TNN-Query is $O(\log \overline{d^+} + k)$. □

5 INDEX CONSTRUCTION

In this section, we first introduce two baselines for index construction in Section 5.1. Then we present a new index construction framework in Section 5.2 followed by the detailed efficient implementation of the framework in Sections 5.3 to 5.5.

5.1 Baselines

From the definition of TNN-Index in Definition 4.1, the construction of TNN-Index is essentially computing the kNNs of each vertex $v \in V$ for each departure time $t_d \in T_d(v)$. Hence, two traversal algorithms can be easily devised to construct the TNN-Index.

Forward Search. A straightforward method for building TNN-Index is to run the INE algorithm (Section 3.1) from each vertex $u \in V$ for each departure time $t_d \in T_d(u)$ to compute the kNN $V_k(u, t_d)$. If u is not an object, we store $V_k(u, t_d)$ in the TNN-Index, otherwise, we store $V_k(u, t_d) \setminus \{(u, t_d)\}$. If two sets $\mathcal{K}(u, t_d)$ and $\mathcal{K}(u, t'_d)$ are identical and $t_d < t'_d$, we remove $\mathcal{K}(u, t_d)$ from the TNN-Index. We refer to this algorithm as Dijk$^{\mathsf{F}}$. Given that the time

Algorithm 2: Naive-Cons-Framework(G)

```
1 $C \leftarrow$ a vertex cut of $G$;
2 $CCs \leftarrow$ connected components of $G \setminus C$;
3 foreach $v \in C$ do                                   // Phase 1
4     Compute $\mathcal{K}(v)$ on $G$;
5 foreach $G_s \in CCs$ do                               // Phase 2
6     foreach $u \in V(G_s)$ do
7         Compute $\mathcal{K}(u)$ with $\{\mathcal{K}(v) | v \in C\}$;
```

complexity for a single execution of the Dijkstra's algorithm is $O((n + m) \cdot \log n)$, the average size of $T_d(v)$ is $O(\overline{d^+})$, hence the time complexity of Dijk$^{\mathsf{F}}$ is $O(n \cdot \overline{d^+} \cdot (n + m) \cdot \log n)$.

Reverse Search. The forward search method requires the execution of the INE algorithm many times, which has a significant computational overhead. To address this issue, another method is to perform reverse Dijkstra searches along the opposite direction of the edges from each object to the vertices in V. Specifically, it constructs the TNN-Index with the following steps: (1) for each object $o \in \mathcal{P}$ and each arrival time $t_a \in \{t_a | \langle v, o, t_d, t_a \rangle \in E\}$, it uses reverse Dijkstra search from o with arrival time t_a to compute the latest departure time t_d for each vertex $v \in V$ such that one can reach o before t_a, then it adds (o, t_a) to $V_k(v, t_d)$; (2) for each $V_k(v, t_d)$ obtained in step (1), it sorts the elements in $V_k(v, t_d)$ by t_a and keeps only the top-k elements; (3) it constructs the TNN-Index using the $V_k(v, t_d)$ obtained in step (2). We refer to this algorithm as Dijk$^{\mathsf{R}}$. Similar to Dijk$^{\mathsf{F}}$, the time complexity of Dijk$^{\mathsf{R}}$ is $O(|\mathcal{P}| \cdot \overline{d^-} \cdot (n + m) \cdot \log n)$.

5.2 A New Index Construction Framework

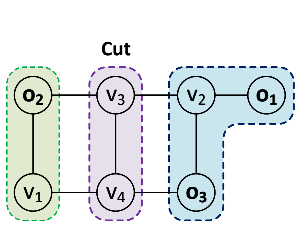
The baseline methods are inefficient as they require traversing the graph many times. To address this issue, we resort to exploiting the planarity property [22, 30] of PTNs and propose a novel index construction framework. In this section, we first introduce the concept of *vertex cut* and the property of kNN problem on graphs, and then present a new index construction framework.

*Definition 5.1 (**Vertex Cut**).* Given a graph G, a vertex set $C \subset V$ is a *vertex cut of* G if deleting C partitions G into multiple disjoint subgraphs. Given two vertices $u, v \in V$, the vertex set C is a *vertex cut of* u *and* v if deleting C from G makes u and v in different connected subgraphs.

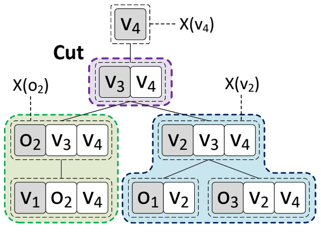
Lemma 5.2. *Given a graph* G*, and a vertex cut* C *of vertices* $u, v \in V$*, any path between* u *and* v *must pass a vertex in* C*.*

Lemma 5.3. *Given a vertex cut* C *separating vertex* u *and an object* o*, if* $(o, t_a) \in V_k(u, t_d)$*, then there must exist* $w \in C$ *and* t'_d *such that* $(o, t_a) \in V_k(w, t'_d)$ *and* $EAT(u, w, t_d) \leq t'_d$*, where* $V_k(\cdot)$ *is defined in Definition 2.3.*

Proof. Assume that there is no $w \in C$ and t'_d such that $(o, t_a) \in V_k(w, t'_d)$ and $EAT(u, w, t_d) \leq t'_d$. With Lemma 5.2, as $(o, t_a) \in V_k(u, t_d)$, there must exists an earliest arrival path from u to o with departure time t_d and arrival time t_a that passes through a vertex $w \in C$. Then we can construct a new kNN $V'_k(u, t_d) = V_k(w, t'_d)$ for u. According to the assumption, it is easy to see that $t_a > \max\{t | (v, t) \in V'_k(u, t_d)\}$, then we have $(o, t_a) \notin V'_k(u, t_d)$, this leads to contradictions, lemma holds. □



(a) Vertex cut



(b) Tree decomposition

Figure 2: Vertex cut property of tree decomposition

The New Index Construction Framework. With Lemma 5.3, a naive index construction framework for constructing the TNN-Index is presented in Algorithm 2. Its idea is very simple. Given a graph G, we first obtain a vertex cut C of G (line 1). Then we compute the kNN indexes $\mathcal{K}(v)$ for each vertex $v \in C$ (line 3-4), we refer to it as the first phase. After that, for each connected component G_s of $G \setminus C$ we compute the kNN indexes $\mathcal{K}(u)$ for each vertex $u \in G_s$ (line 5-7), we refer to it as the second phase. During the second phase, the search space is limited to G_s with the help of kNN indexes of the cut C obtained in the first phase. The correctness of the framework is guaranteed by Lemma 5.3.

The advantage of the Naive-Cons-Framework lies in the utilization of the vertex cut C for reducing the search space. The naive framework is inefficient since it requires conducting searches in the subgraphs. To further optimize it, we can apply this naive framework recursively on the connected components G_s in $G \setminus C$ until we reach the base case where G_s is a single vertex or a clique, and the search space is reduced to a very small size.

Example 5.4. Figure 2 (a) examplifies a vertex cut $C = \{v_3, v_4\}$ of the graph in Figure 1. To create the TNN-Index for G, we first compute the kNN indexes $\mathcal{K}(v_3)$ and $\mathcal{K}(v_4)$ on G. After removing the cut C, there are two connected components: $G_1 = \{v_1, o_2\}$ and $G_2 = \{o_1, v_2, o_3\}$. We then compute the kNN indexes for each vertex in the two connected components. If we apply the Naive-Cons-Framework recursively on G_2, the cut is $C_2 = \{v_2\}$. We can first calculate $\mathcal{K}(v_2)$ within G_2 then the kNN indexes for the remaining vertices in G_2. In fact, the kNN indexes of the vertices in C_2 on G_2 can be used to accelerate the computation of the kNN indexes of the vertices in C by avoiding visiting o_1.

Organization of the subsequent sections. In the subsequent sections, we will cover the details of the recursive version of the Naive-Cons-Framework. In particular, we will introduce the notion of tree decomposition in Section 5.3, which helps to obtain the vertex cuts in a hierarchical tree structure. Then in Section 5.4, we will present the algorithm for the EP tree decomposition, which helps to avoid the Dijkstra searches in the connected components. Finally, in Section 5.5, we will present the final two-phase index construction algorithm for the TNN-Index, which is the recursive version of the Naive-Cons-Framework, and the two phases in the final algorithm correspond to the two non-recursive phases in the Naive-Cons-Framework.

5.3 Tree Decomposition

Tree decomposition [25], a technique to map a graph into a tree, can enhance the efficiency of solving graph-related computational problems with its vertex cut property. Given a graph G, a tree decomposition of it is defined as follows [3]:

*Definition 5.5 (**Tree Decomposition**).* The tree decomposition T_G of a directed graph $G(V, E)$ is a rooted tree. For each $v \in V(G)$, there is a node $X(v) \in V(T_G)$ containing a subset of vertices in G, i.e., $X(v) \subseteq V(G)$. T_G meets the following three conditions:

(1) $\bigcup_{X \in V(T_G)} X = V(G)$;
(2) For each edge $e(u, v) \in E(G)$ in the graph G, there exists a node X in $V(T_G)$ such that $u \in X$ and $v \in X$;
(3) For any vertex $v \in G$, $\{X | v \in X\}$ forms a connected subtree, which is rooted at $X(v)$.

In the remainder of this paper, we refer to every $v \in V(G)$ in the graph G as a *vertex* and every $X \in V(T_G)$ in the tree T_G as a *node*.

*Definition 5.6 (**Treewidth and Treeheight**).* Given a tree decomposition T_G of graph G, the *treewidth* $tw(T_G)$ is 1 less than the maximum size of all nodes in T_G, i.e., $tw(T_G) = \max_{X \in V(T_G)} |X| - 1$. The *treeheight* $th(T_G)$ is the maximum depth of all nodes in T_G, where the depth of a node is the shortest distance from that node to the root node. The treewidth of G is the minimum treewidth of all tree decompositions of G.

Properties of Tree Decomposition. Below we present the vertex cut properties of the tree decomposition.

Lemma 5.7. *Given two nodes $X(s), X(t) \in T_G$, if they don't have ancestor/descendant relationship, the lowest common ancestor (LCA) X_{lca} of $X(s)$ and $X(t)$ is the vertex cut of G and separates s and t [4].*

Lemma 5.8. *For any two nodes $X_c, X_p \in T_G$, X_c is the child of X_p and X_p is the parent of X_c, if there exists $s \in X_c \setminus X_p$ and $t \in X_p \setminus X_c$, then $X_c \cap X_p$ is a vertex cut of G and separates s and t.*

Proof. We assume vertex set $C = X_c \cap X_p$ is not a vertex cut. If we cut the edge (X_c, X_p) of the tree, then T_G breaks into two subtrees, T_c (containing X_c) and T_p (containing X_p). By Condition 3 of Definition 5.5, it is easy to see that $V_c = V(T_c) \setminus C$ and $V_p = V(T_p) \setminus C$ are two disjoint sets. Since C is not a vertex cut, then the $V_c \cup V_p$ induced subgraph $G_{[V_c \cup V_p]}$ is a connected graph, so there must exist an edge (u, v) in $G_{[V_c \cup V_p]}$ such that $u \in V_c$ and $v \in V_p$. However, it is easy to see that (u, v) doesn't belong any tree node, which contradicts condition 2 of Definition 5.5. □

Example 5.9. Figure 2 (a) is the graph preserving only the topology of the graph G in Figure 1 and Figure 2 (b) is a tree decomposition T_G of G. Given a vertex v_2 in G, the node $X(v_2)$ is marked in T_G. The treewidth $tw(T_G)$ is 2 while the treeheight $th(T_G)$ is 4. Node $X(o_2)$ is the parent of $X(v_1)$, the intersection $X(v_1) \cap X(o_2) = \{o_2, v_4\}$ is a vertex cut of G that separates v_1 and v_3. The LCA of $X(v_1)$ and $X(v_2)$, $X(v_3) = \{v_3, v_4\}$, is also a vertex cut of G.

5.4 Algorithm for EP Tree Decomposition

Based on the definition of treewidth, the size of the vertex cuts obtained in tree decomposition is bounded by the treewidth. A smaller vertex cut aids in reducing the computation cost. Given a graph G, determine if G has a tree decomposition with treewidth less than a variable is NP-complete [1]. Therefore, heuristics are

Algorithm 3: EP-TreeDecomposition($G(V,E)$)

```
1  $G_0 \leftarrow G$; $T_G \leftarrow \emptyset$;
2  for $i \leftarrow 1$ to $n$ do
3      $v \leftarrow \arg\min_{v \in V(G_{i-1})} |N(v, G_{i-1})|$;
4      $\pi(v) \leftarrow i$;
5      create node $X(v) = \{v\} \cup N(v, G_{i-1})$;
6      add $X(v)$ to $T_G$;
7      for $u \in N_{out}(v, G_{i-1})$ do
8          $\mathsf{E_p}(v,u) \leftarrow$ comput the edge profile of $E(v,u,G_{i-1})$
9      for $u \in N_{in}(v, G_{i-1})$ do
10         $\mathsf{E_p}(u,v) \leftarrow$ comput the edge profile of $E(u,v,G_{i-1})$
11     $G_i \leftarrow G_{i-1} \setminus \{v\}$;
12     for $\langle u, v, t_d, t_a \rangle \in E_{in}(v, G_{i-1})$ do
13         for $\langle v, w, t'_d, t'_a \rangle \in E_{out}(v, G_{i-1})$ s.t. $w \neq u$ do
14             if $t_a \le t'_d$ then
15                 add $\langle u, w, t_d, t'_a \rangle$ to $G_i$
16     foreach $u, w \in N(v, G_{i-1})$ where $u \neq w$ do
17         if $u \notin N(w, G_{i-1})$ then
18             connect $u$ and $w$ by adding $\langle u, w, -\infty, +\infty \rangle$ to $G_i$;
19 foreach $v \in V(G)$ do
20     if $\pi(v) < n$ then
21         $u \leftarrow$ vertex in $X(v) \setminus \{v\}$ with smallest $\pi(\cdot)$ value;
22         set $X(u)$ as the parent node of $X(v)$ in $T_G$;
23 return $T_G$;
```

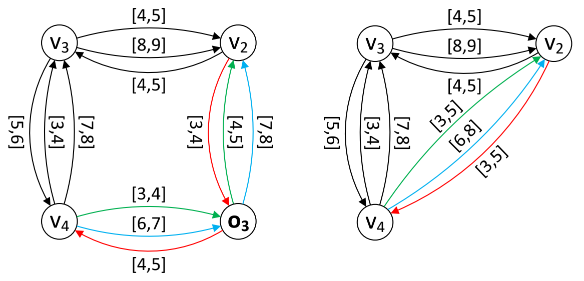
used to find T_G with small treewidth [32]; a popular one is the minimum degree elimination (MDE) [2].

MDE for directed multi-graphs. The MDE algorithm is initially designed for undirected simple graphs [2]. Here we extend it to directed multi-graphs. Given a directed multi-graph G, we first eliminate the vertex v in G with the fewest neighbors, and add edges between every pair of v's neighbors such that $N(v)$ forms a clique in G, i.e., for each vertex pair u, w from $N(v)$, we have $u \in N(w)$ and $w \in N(u)$. Then v and its neighbors $N(v)$ are collected as a node $X(v)$ in the tree. We repeats the above process until all vertices are eliminated. After that, for each node $X(v)$, we set a node $X(u)$ as its parent node, where $X(u)$ is the node created by the first eliminated vertex u in $X(v) \setminus v$.

EP Vertex Elimination. In the MDE algorithm, only the topology of the graph is considered while the multi-edges and their time intervals are ignored. To facilitate the efficient computation of the kNN index, we propose the *EAT-Preserved (EP) Vertex Elimination*, which preserves the earliest arrival time of the graph:

Given a graph G and a vertex $v \in V(G)$, the *EP Vertex Elimination* for v in G transforms G into another graph as follows: for each edge $\langle u, v, t_d, t_a \rangle$ in $E_{in}(v)$ and each edge $\langle v, w, t'_d, t'_a \rangle$ in $E_{out}(v)$, if $t_a \le t'_d$, we add a new edge $\langle u, w, t_d, t'_a \rangle$ to G. Then we check if every pair of vertices (u, v) in $N(v)$ are connected by an edge, if not, we add a dummy edge $\langle u, w, -\infty, +\infty \rangle$ between them to make them neighbors. Finally, we eliminate v and its incident edges from G. We denote the graph after EP Vertex Elimination as G'.

*Definition 5.10 (**EP-Graph**).* Given two graphs G and G' with $V(G') \subseteq V(G)$, for any $u, v \in V(G')$ and any departure time t_d, if $EAT_{G'}(u, v, t_d) = EAT_G(u, v, t_d)$, then we say G' is an *EAT-*Preserved Graph (EP-Graph) of G and we denote it as $G' \sqsubseteq G$.



(a) Before eliminating o_3 **(b) After eliminating o_3**

Figure 3: Example of EP vertex elimination

LEMMA 5.11. *Given a graph G and a vertex $v \in V(G)$, if G' is the graph after EP Vertex Elimination for v in G, then G' is an EP-Graph of G, i.e., $G' \sqsubseteq G$.*

PROOF. Given any $u, w \in V(G')$ and any departure time t_d, for the corresponding path of $EAT_G(u, w, t_d)$, we consider two cases:

Case 1: the path does not contain v. Then the path is also a path in G', so $EAT_{G'}(u, w, t_d) = EAT_G(u, w, t_d)$.

Case 2: the path contains v. Then the path must contain an edge $\langle u, v, t_d, t_a \rangle$ in $E_{in}(v)$ and an edge $\langle v, w, t'_d, t'_a \rangle$ in $E_{out}(v)$, where $t_a \le t'_d$. By the definition of EP Vertex Elimination, we add a new edge $\langle u, w, t_d, t'_a \rangle$ to G', so $EAT_{G'}(u, w, t_d) = EAT_G(u, w, t_d)$. □

The EP vertex elimination adds many edges to the graph, some of them are redundant. Hence, we introduce the concept of *Edge Profile* to eliminate the redundant edges.

*Definition 5.12 (**Edge Dominance**).* Given two edges $\langle u, v, t_d, t_a \rangle$ and $\langle u, v, t'_d, t'_a \rangle$, if $t_d \ge t'_d$ and $t_a \le t'_a$, then we say $\langle u, v, t_d, t_a \rangle$ *dominates* $\langle u, v, t'_d, t'_a \rangle$.

*Definition 5.13 (**Edge Profile**).* Given a set of edges $E(u, v)$ from u to v, then we define the *Edge Profile*, denoted as $\mathsf{E_p}(u, v)$, as the set of edges in $E(u, v)$ that are not dominated by other edges.

Example 5.14. Figure 3 demonstrates the EP vertex elimination for o_3. The set of edges between $N(o_3)$, v_4 and v_2, in Figure 3 (b) is the edge profile of the edges generated during the elimination process. The red edge $\langle v_2, v_4, 3, 5 \rangle$ is generated by connecting the edge $\langle v_2, o_3, 3, 4 \rangle$ and the edge $\langle o_3, v_4, 4, 5 \rangle$.

EP Tree Decomposition. Algorithm 3 presents the overall process of the EP tree decomposition. Given a graph G, we select the vertex v with the smallest number of neighbors to eliminate (line 3) and use $\pi(v)$ to record the elimination order of v (line 4), where $\pi(v)$ will be used to determine the parent-child relationship between tree nodes. When eliminating a vertex, we first create a node $X(v)$ containing v and its neighbors and add the node to T_G (line 5-6). Then we compute the edge profile and remove the redundant edges between v and its neighbors (line 7-10). Next, we apply EP Vertex Elimination to remove v and its incident edges from G to form G_i (line 11-18). After all the vertices are eliminated, for each vertex v, we find the vertex $u \in X(v) \setminus \{v\}$ with the smallest value of $\pi(u)$, and make the corresponding $X(u)$ the parent node of $X(v)$ (Line 19-22). As a result, the tree nodes $X(v)$ with larger $\pi(v)$ will be on the upper part of the tree and the tree node with the largest $\pi(v)$ will be the tree root.

LEMMA 5.15. *For any G_i generated in Algorithm 3, $G_i \sqsubseteq G$.*

Theorem 5.16. *The running time of Algorithm 3 is bounded by* $O(n \cdot (\log n + tw^2 \cdot \overline{d^-} \cdot \overline{d^+})))$.

Proof. For each loop, finding the vertex with the smallest number of neighbors takes $O(\log n)$ time. For each EP vertex elimination, the dominant cost is to create the clique in line 12-18, which takes $O(tw^2 \cdot \overline{d^-} \cdot \overline{d^+})$ time. Hence, the total time complexity is $O(n \cdot (\log n + tw^2 \cdot \overline{d^-} \cdot \overline{d^+})))$. □

Example 5.17. Figure 2 (b) is the tree decomposition of the graph in Figure 1 generated by Algorithm 3.

5.5 Two-Phase Index Construction Algorithm

Our proposed index construction algorithm relies on the edge profiles that are generated during the EP tree decomposition of a PTN. Before introducing the details of the algorithm, we first introduce the Operator Join $\odot$ and the Operator Union $\sqcup$ which will be used in the index construction algorithm.

Operator Join $\odot$. Given $\mathsf{E_p}(u,v)$ and $\mathcal{K}(v)$, the Join operator $\odot$ enables u to acquire kNNs from its neighbor v. The basic procedure involves matching $(t'_d, t'_a) \in \mathsf{E_p}(u,v)$ and $\mathcal{K}(v, t_d) \in \mathcal{K}(v)$ with $t'_a \le t_d$ to obtain $\mathcal{K}(u)$. Specifically, for each $\mathcal{K}(v, t_d) \in \mathcal{K}(v)$, we examine $\mathsf{E_p}(u,v)$ to identify the **latest possible departure time** t'_d such that $(t'_d, t'_a) \in \mathsf{E_p}(u,v)$ and $t'_a \le t_d$. If such a time exists, we can establish a kNN entry for u from v, where $\mathcal{K}(u, t'_d) = \mathcal{K}(v, t_d)$. In cases where no such time exists, e.g., no journey from u to v arrives before t_d, then $\mathcal{K}(v, t_d)$ is discarded by u. By identifying the latest possible departure time from $\mathsf{E_p}(u,v)$ for each $\mathcal{K}(v, t_d)$, the redundancy like $\mathcal{K}(u, t_{d1}) = \mathcal{K}(u, t_{d2})$ and $t_{d1} < t_{d2}$ will not occur. Because we only keep t_{d2} which is the latest possible departure time for $\mathcal{K}(u, t_{d2})$ and t_{d1} is omitted.

Operator Union $\sqcup$. Given $\mathcal{K}_1(u)$ and $\mathcal{K}_2(u)$ of u that are obtained from different sources, e.g., two neighbors of u , the operator Union $\sqcup$ helps u to calculate its aggregated kNN index $\mathcal{K}(u)$ by merging $\mathcal{K}_1(u)$ and $\mathcal{K}_2(u)$. This process has two steps:

Step 1. Merge. In the first step, we directly merge items in $\mathcal{K}_1(u)$ and $\mathcal{K}_2(u)$ and obtain a tentative $\mathcal{K}(u)$. Specifically, if a departure time t_d only exists in one of $\mathcal{K}_1(u)$ and $\mathcal{K}_2(u)$, the respective $\mathcal{K}_1(u, t_d)$ or $\mathcal{K}_2(u, t_d)$ is assigned to $\mathcal{K}(u, t_d)$. Otherwise, if t_d appears in both $\mathcal{K}_1(u)$ and $\mathcal{K}_2(u)$, we keep top-k objects from them in $\mathcal{K}(u, t_d)$ based on their *EAT*.

Step 2. Compensation. Given a departure time t_d from u, if we don't depart immediately but wait a little while for another bus, we may reach some objects with earlier arrival times or have more choices of objects. For instance, if $\mathcal{K}(u,3) = \{(o_1,6),(o_3,10)\}$ and $\mathcal{K}(u,6) = \{(o_2,9),(o_1,11)\}$, then $(o_2,9)$ should be inserted to $\mathcal{K}(u,3)$. Such situation occurs when merging $\mathcal{K}(u)$ from different neighbours. In order to cope with the above situation, we need to use the kNN with a later departure time to update the kNN with a earlier departure time. This can be efficiently implemented by a linear scan of $\mathcal{K}(u)$ from the latest to the earliest departure time.

Example 5.18. To illustrate the above operators, we take the procedure of Phase 1 in Table 3 as an example. The blue dotted line in the graph from v_3 to v_2 indicates the $\mathsf{E_p}(v_3, v_2)$. The second row of the equation demonstrates the process of $\mathsf{E_p}(v_3, v_2) \odot \mathcal{K}(v_2)$ which returns the kNNs of v_3 obtained from v_2. In particular, $\mathcal{K}(v_2, 3)$ is discarded because $t_d = 3$ cannot be caught up by any edge in $\mathsf{E_p}(v_3, v_2)$. Subsequently, $t_d = 5$ and 9 are connected with edge $\langle v_3, v_2, 4, 5\rangle$ and $\langle v_3, v_2, 8, 9\rangle$, respectively. By replacing with new departure times, we obtain the result, denoted as $\mathcal{K}'(v_3)$.

The third row of the equation demonstrates the process of $\mathcal{K}(v_3) \sqcup \mathcal{K}'(v_3)$. The items with the same departure time are marked in yellow. The first step of $\sqcup$ operator is to directly merge these items, the items in the merge result are marked with colors. We then do the second step of $\sqcup$ operator. Specifically, we add $(o_1, 11)$ from $V_k(v_3, 8)$ to $V_k(v_3, 5)$, which is indicated by the green arrow. Similarly, we add $(o_2, 6)$ from $V_k(v_3, 5)$ to $V_k(v_3, 4)$. Finally, we obtain the final kNN index $\mathcal{K}(v_3)$ of v_3 in the graph.

Algorithm 4: Index-Construction($G(V,E)$)

```
1  $T_G \leftarrow$ EP-TreeDecomposion($G$);
2  foreach $v \in V(G)$ do $\mathcal{K}(v) \leftarrow \emptyset$;
   // Phase 1
3  for $i \leftarrow 1$ to $n$ do
4      $v \leftarrow$ vertex with $\pi(\cdot) = i$;
5      for $u \in X(v) \setminus \{v\}$ do
6          $\mathcal{K}(u) \leftarrow \mathcal{K}(u) \sqcup (\mathsf{E_p}(u,v) \odot \mathcal{K}(v))$;
7          if $v \in \mathcal{P}$ then
8              $\mathcal{K}(u) \leftarrow \mathcal{K}(u) \sqcup$
                 $\{(t_d, \{(v, t_a)\}) | (t_d, t_a) \in \mathsf{E_p}(u,v)\}$;
   // Phase 2
9  for $i \leftarrow n$ to 1 do
10     $v \leftarrow$ vertex with $\pi(\cdot) = i$;
11     for $u \in X(v) \setminus \{v\}$ do
12         $\mathcal{K}(v) \leftarrow \mathcal{K}(v) \sqcup (\mathsf{E_p}(v,u) \odot \mathcal{K}(u))$;
13         if $v \in \mathcal{P}$ then
14             $\mathcal{K}(v) \leftarrow \mathcal{K}(v) \setminus \{v\}$;
15         if $u \in \mathcal{P}$ then
16             $\mathcal{K}(v) \leftarrow \mathcal{K}(v) \sqcup \{(t_d, \{(u, t_a)\}) | (t_d, t_a) \in$
                 $\mathsf{E_p}(v,u)\}$;
17 return TNN-Index;
```

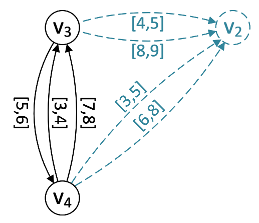
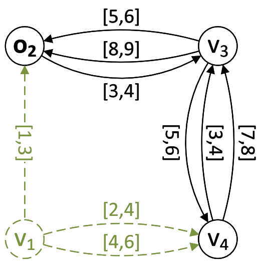
Two-Phase Index Construction Algorithm. Now we are ready to present the TNN-Index construction algorithm, which is presented in Algorithm 4. Given a graph G, the algorithm first computes the EP tree decomposition T_G of G (line 1). Then it constructs the TNN-Index of G in two phases: (1) computing the kNN in subgraphs in a bottom-up manner (lines 3-8); (2) computing the kNN in the entire graph in a top-down manner (Lines 9-16). Next, we introduce the details of the two phases:

• *Phase 1. Bottom-Up kNN Computation on Subgraphs.* This phase is to compute the kNN of each vertex v in a subgraph, it corresponds to Phase 1 of the framework in Section 5.2, where the vertex v acts as a "cut vertex" in the subgraph. Specifically, for each vertex v in the vertex elimination order (line 3-4), we use $\mathcal{K}(v)$ to update the kNN of each vertex u in $X(v) \setminus \{v\}$ (line 5-8). To achieve this, we first use $\mathsf{E_p}(u,v) \odot \mathcal{K}(v)$ to transfer the kNN of v to u through edges from u to v with the edge profile obtained in Algorithm 3, then use $\mathcal{K}(u) \sqcup (\mathsf{E_p}(u,v) \odot \mathcal{K}(v))$ to update the kNN of u (line 6). If v is an object, then we also need to add v to the kNN of u (line 8).

Lemma 5.19. *For any vertex v in G, once Phase 1 is completed, let $S = \{u | X(u) \in T(v)\}$, where $T(v)$ is the subtree of T_G rooted at $X(v)$, then $\mathcal{K}(v)$ is the* TNN-Index *of v on the subgraph $G[S]$.*

Proof. We prove this lemma by induction. First, if $X(v)$ is a leaf node, then the subgraph only contains v, so the lemma holds

Table 3: Examples of Phase 1 and Phase 2

	Current EP-graph	kNN Computation
Phase 1	 v3 v2 v4 [4,5] [8,9] [3,5] [6,8] [5,6] [3,4] [7,8]	$\mathcal{K}(\mathrm{v}_3) \leftarrow \mathcal{K}(\mathrm{v}_3) \sqcup (\mathsf{E}_\mathrm{p}(v_3, v_2) \odot \mathcal{K}(\mathrm{v}_2))$ $= \mathcal{K}(\mathrm{v}_3) \sqcup \left(\begin{Bmatrix} (4,5), \\ (8,9) \end{Bmatrix} \odot \begin{Bmatrix} 3:\{(\mathrm{o}_3,4),(\mathrm{o}_1,8)\}, \\ 5:\{(\mathrm{o}_1,8)\}, \\ 9:\{(\mathrm{o}_1,11)\} \end{Bmatrix} \right)$ $= \begin{Bmatrix} 5:\{(\mathrm{o}_2,6)\}, \\ 8:\{(\mathrm{o}_2,9)\} \end{Bmatrix} \sqcup \begin{Bmatrix} 4:\{(\mathrm{o}_1,8)\}, \\ 8:\{(\mathrm{o}_1,11)\} \end{Bmatrix} = \begin{Bmatrix} 4:\{(\mathrm{o}_2,6),(\mathrm{o}_1,8)\}, \\ 5:\{\overline{(\mathrm{o}_2,6)},(\mathrm{o}_1,11)\}, \\ 8:\{(\mathrm{o}_2,9),\overline{(\mathrm{o}_1,11)}\} \end{Bmatrix}$
Phase 2	 o2 v3 v1 v4 [5,6] [8,9] [3,4] [1,3] [2,4] [4,6] [5,6] [3,4] [7,8]	$\mathcal{K}(\mathrm{v}_1) \leftarrow \mathcal{K}(\mathrm{v}_1) \sqcup (\mathsf{E}_\mathrm{p}(v_1, o_2) \odot \mathcal{K}(\mathrm{o}_2))$ $= \emptyset \sqcup \left(\{(1,3)\} \odot \{3:\{(\mathrm{o}_3,7),(\mathrm{o}_1,8)\}\} \right) = \{1:\{(\mathrm{o}_3,7),(\mathrm{o}_1,8)\}\}$ $\mathcal{K}(\mathrm{v}_1) \leftarrow \mathcal{K}(\mathrm{v}_1) \sqcup \{(t_d,\{(v_1,t_a)\}) \mid (t_d,t_a) \in \mathsf{E}_\mathrm{p}(v_1,o_2)\}$ $= \mathcal{K}(\mathrm{v}_1) \sqcup \{1:\{(\mathrm{o}_2,3)\}\} = \{1:\{(\mathrm{o}_2,3),(\mathrm{o}_3,7),(\mathrm{o}_1,8)\}\}$ $\mathcal{K}(\mathrm{v}_1) \leftarrow \mathcal{K}(\mathrm{v}_1) \sqcup (\mathsf{E}_\mathrm{p}(v_1, v_4) \odot \mathcal{K}(\mathrm{v}_4))$ $= \mathcal{K}(\mathrm{v}_1) \sqcup \left(\begin{Bmatrix} (2,4), \\ (4,6) \end{Bmatrix} \odot \begin{Bmatrix} 3:\{(\mathrm{o}_3,4),(\mathrm{o}_2,6),(\mathrm{o}_1,8)\}, \\ 6:\{(\mathrm{o}_3,7),(\mathrm{o}_2,9),(\mathrm{o}_1,11)\}, \\ 7:\{(\mathrm{o}_2,9),(\mathrm{o}_1,11)\} \end{Bmatrix} \right) = \begin{Bmatrix} 1:\{(\mathrm{o}_2,3),(\mathrm{o}_3,7),(\mathrm{o}_1,8)\}, \\ 4:\{(\mathrm{o}_3,7),(\mathrm{o}_2,9),(\mathrm{o}_1,11)\} \end{Bmatrix}$

trivially. Second, if $X(v)$ is not a leaf node , and we suppose the lemma holds for all vertices whose corresponding nodes are descendants of $X(v)$, then we prove it for v. With EP-tree decomposition, for any $u \in C(v)$, $G_{\pi(u)-1} \sqsubseteq G$, it's not hard to know that v's kNN on $G[T(u)]$ can be obtained from u via $\mathsf{E}_\mathrm{p}(v,u)$, i.e., $\mathsf{E}_\mathrm{p}(v,u) \odot \mathcal{K}(u) \sqcup \{(t_d,\{(v,t_a)\}) \mid (t_d,t_a) \in \mathsf{E}_\mathrm{p}(v,u)\}$. Let $C(v)$ denote the vertices u whose corresponding nodes $X(u)$ are the descendant of $X(v)$ in T_G and contain v, i.e., $C(v) = \{u \mid v \in X(u) \wedge X(u)$ is a descendant of $X(v)\}$. Then $C(v)$ is the vertex cut of G if $C(v) \cup \{v\}$ doesn't contain all the vertices in S [23]. Therefore, after all the vertices in $C(v)$ are eliminated, the $\mathcal{K}(v)$ is the TNN-Index of v on $G[S]$. The lemma holds by induction. □

Example 5.20. The first row in Table 3 demonstrates the steps in phase 1 for updating $\mathcal{K}(v_3)$ with $\mathcal{K}(v_2)$ with aforementioned operators. The final result is the updated $\mathcal{K}(v_3)$ by merging the kNNs obtained from v_2.

• *Phase 2. Top-Down kNN Computation on the Entire Graph.* This phase corresponds to Phase 2 of the framework in Section 5.2 and is presented in lines 9-16 of Algorithm 4. We compute the kNN of vertices following the reverse order of vertex elimination (lines 9-10). Specifically, for each vertex v in the reverse order of vertex elimination, we use the $\mathcal{K}(u)$ of each vertex u in $X(v) \setminus \{v\}$ to update the $\mathcal{K}(v)$ (lines 11-12). If v is an object, then v could be a kNN of u and it could be added to the kNN of itself, so we remove v from the $\mathcal{K}(v)$ (lines 13-14) since we don't include an object itself in its index. While if u is an object, we have to add u to $K(v)$ with line 16 as $\mathcal{K}(u)$ doesn't contain itself. This phase completes when all the vertices are processed.

Lemma 5.21. *For any vertex v in G, once Phase 2 is completed, then $\mathcal{K}(v)$ is the* TNN-Index *of v on the entire graph G.*

Proof. We prove this lemma by induction. First, if $\pi(v) = n$, then v is the last vertex to be eliminated, so the lemma holds according to Lemma 5.19. Second, if $\pi(v) < n$, assume the lemma holds for all vertices $u \in X(v) \setminus \{v\}$, then we prove it for v. If the conditions hold for Lemma 5.8, $X(v) \setminus \{v\}$ is the vertex cut of G, then by Lemma 5.3 and Lemma 5.19, then $\mathcal{K}(v)$ holds the kNNs of v on the entire graph G. Otherwise, $X(v) \cup \{w \in X(u) \mid X(u)$ is a descendant of $X(v)\}$ contains all the vertices in G, then by Lemma 5.19, it is easy to know that then $\mathcal{K}(v)$ holds the kNNs of v on the entire graph G. Hence this lemma holds by induction. □

Example 5.22. The second row in Table 3 outlines the process of updating $\mathcal{K}(v_1)$ with $\mathcal{K}(o_2)$ and $\mathcal{K}(v_4)$ during Phase 2. The green dotted lines indicates $\mathsf{E}_\mathrm{p}(v_1,o_2)$ and $\mathsf{E}_\mathrm{p}(v_1,v_4)$. The first equation shows of calculation of $\mathcal{K}(v_1)$ with $\mathcal{K}(o_2)$. Given that $o_2 \in \mathcal{P}$, an entry $(o_2, 3)$ with $t_d = 1$ is added, based on $\mathsf{E}_\mathrm{p}(v_1,o_2)$, as illustrated in the second equation. Following this, v_1 acquires kNNs from v_4. For $\mathsf{E}_\mathrm{p}(v_1,v_4) \odot \mathcal{K}(v_4)$, the edge $\langle v_1, v_4, 2, 4\rangle$ becomes redundant due to the existence of edge $\langle v_1, v_4, 4, 6\rangle$ when joined with the kNN with $t_d = 6$, as kNN with departure time 2 is identical to that with departure time 4. The final result of $\mathcal{K}(v_1)$ is the index of v_1 in G.

Theorem 5.23. *The time cost of Algorithm 4 is $O(n \cdot tw \cdot \overline{d^+} \cdot k)$.*

Proof. Each vertex v uses its kNN index to update the indexes of the vertices $u \in X(v)$, where $|X(v)|$ is limited by treewidth tw. The $\odot$ operation scans $\mathsf{E}_\mathrm{p}(u,v)$ and $\mathcal{K}(v)$ linearly. Therefore, the time cost of $\odot$ is $O(\overline{d^+} \cdot k)$. The $\sqcup$ operation, updating u's kNN, involves linear scanning and merging of the operands starting from the largest t_d. In the worst case, merging kNN at the same t_d with later times costs $O(k+k+k)$, leading to a time complexity of $O(\overline{d^+} \cdot k)$ for $\sqcup$. Hence, total time complexity for Phase 1 is $O(n \cdot tw \cdot \overline{d^+} \cdot k)$. The time complexity of Phase 2 is similar to Phase 1. Hence, the total time complexity of Algorithm 4 is $O(n \cdot tw \cdot \overline{d^+} \cdot k)$. □

6 kNN SEARCH WITH OPENING HOURS

Facilities in a city like shops or restaurants, may not always be open. Users using kNN queries might find some of them closed upon their arrival. For example, if the opening hours of a pharmacy are from 9 am to 4 pm. If a user can arrive at this shop at 8 am but needs to wait until 8 for access. This raises an important question: should the user wait for it to open or choose an alternative that the user can arrive at 8:20 am but is already open? If we consider the accessibility of the store, the latter should be a better choice. Assume the opening hours of an object v is $T^o(v) = \{[t_1^o, t_1^c], [t_2^o, t_2^c], ..., [t_n^o, t_n^c]\}$. We aim to help users find top-k objects that are accessible at the earliest time. A straightforward method is to adapt the INE algorithm for this problem. Another method is to update the TNN-Index each time an object is opened or closed. However, updating the index cannot guarantee a correct response as an object may be closed during the journey to it. To address these issues, we strive to encode the opening hours of objects into the index structure. First, given the opening hours $T^o(v)$ of an object v and the earliest arrival time t_{ea} of v, we define the *accessible time* of v as follows:

$$T^a(v) = T^o(v) \cap [t_{ea}, \infty] \quad (1)$$

If $T^a(v)$ is empty, which means is not accessible, then we should ignore this object. Otherwise, the earliest accessible time of v is obtained with the following equation:

$$t^a = \min\{t^o | (t^o, t^c) \in T^a(v)\} \quad (2)$$

To encode the opening hours into the index, we can simply replace the *EAT* with the earliest accessible time in the index.

7 INDEX COMPRESSION

We observed that for certain consecutive departure times of a vertex in TNN-Index, their kNNs might consist of the same set of objects in identical order. For instance, the $\mathcal{K}(v_4, 3)$ and $\mathcal{K}(v_4, 6)$ in Table 2 consist of the same object set $\{o_3, o_2, o_1\}$ but with varying *EAT*s. In such scenarios, we can retain just a single copy of the object set, while documenting the *EAT*s for each t_d separately. This results in a more compact version of TNN-Index, referred to as TNNC-Index. Table 4 illustrates the TNNC-Index of v_4. For $\mathcal{K}(v_4, 3)$ and $\mathcal{K}(v_4, 6)$, we store a single copy of the object set $\{o_3, o_2, o_1\}$, while their earliest arrival times are stored separately.

Table 4: TNNC-Index of v_4

v	t_d	object set	*EATs*
v_4	3	o_3, o_2, o_1	4, 6, 8
	6		7, 9, 11
	7	o_2, o_1	9, 11

8 EXPERIMENTS

Settings. Experiments are conducted on a Linux server with Intel Xeon Gold 6342 CPU and 64GB memory. The algorithms are implemented in C++ and compiled with GCC 11.4.0.

Methods. We evaluate the performance of the following methods:

- INE, INE^o [18]: The Dijkstra-based online search method (Section 3.1) and its extension for scenarios with opening hours.
- TFS, TFS^o [14]: An index-based method (Section 3.2) and its extension for scenarios with opening hours.
- PTLDB, PTLDB^o [10]: Another index-based method (Section 3.2) and its extension for scenarios with opening hours.
- TD-GLAD, TD-GLAD^o [17]: An index-based method with distance bounded search for time-dependent graphs (Section 3.2) and its extension for scenarios with opening hours.
- TNN, TNN^o: Our proposed method and its extension for scenarios with opening hours.

Table 5: Network Statistics

Dataset	Region	|V|	|E|	*tw*
SLC	Salt Lake City	5,278	845,855	22
SYD	Sydney	43,790	4,394,307	130
CHI	Chicago	11,032	4,938,175	69
SE	Sweden	49,018	9,514,878	43
NO	Norway	89,922	7,477,992	64
CH	Swiss	38,457	16,073,351	107
BA	Buenos Aires	44,621	28,024,159	311
UK	UK	330,177	68,377,836	246
FLA	Florida	1,070,376	226,340,315	94
LKS	Great Lakes	2,758,119	323,543,821	264

Table 6: Parameters and their settings

Parameter	Values
k: number of nearest neighbors	10, **20**, 30, 40, 50
object density: $\|\mathcal{P}\|/\|V\|$	1‰, **5‰**, 1%, 5%, 10%

- TNNC, TNNC^o: Index compressed version of TNN and TNN^o.
- Dijk^F, Dijk^R: Two baselines for index construction (Section 5.1).

Datasets. We use 8 real-world public transportation networks downloaded from TransitFeeds[1] and 2 synthetic networks (FLA and LKS), which are generated based on the road networks from 9th DIMACS Implementation Challenge[2]. For the synthetic networks, we retain the vertices from the original graphs, for each edge in the original graph, we replace it with a set of edges with random departure and arrival times. These sets of edges simulate real-world scenarios by featuring high frequency during the day and low frequency at night. The details of these datasets are presented in Table 5. The initial two columns provide the dataset's notation and respective regions. This is followed by the number of vertices, edges, and the treewidth.

Query Sets. We randomly select 1,000 vertices for each dataset to serve as query vertices. For every selected vertex, query departure times are set from 7:00 to 21:00, at intervals of 20 minutes.

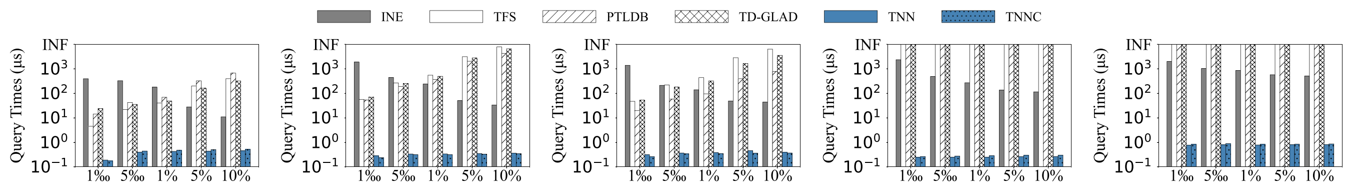
Parameter Settings. Objects are generated randomly for each dataset. Table 6 lists the selected query parameter k and object density for our experiments, with the defaults in bold. For experiments with opening hours, each object is randomly assigned one of three opening timeframes: open 24 hours, open from 9:00 to 17:00, or open during the slots [8:00, 11:00] and [13:00, 16:00].

8.1 Query Processing

Exp-1: Query time by varying object density. We evaluate the query processing time of different methods by varying object densities. For each dataset, five groups of objects are generated with density = {1‰, 5‰, 1%, 5%, 10%} and the average query processing time for each algorithm is reported in Figure 4.

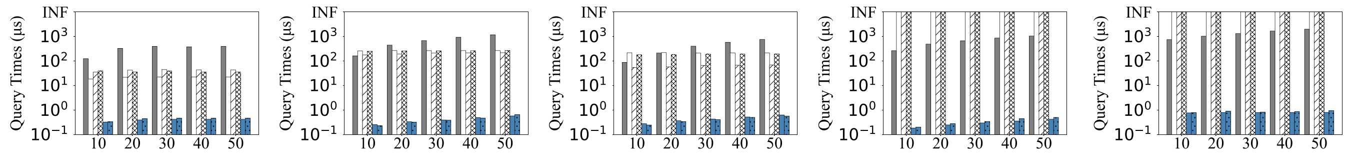
For the large datasets BA, UK, FLA and LKS, path index-based methods TFS, PTLDB and TD-GLAD failed to construct an index in eight hours. From the result, as the object density increases, the query time for INE decreases, while the query time for the path index-based methods increases. TD-GLAD outperforms TFS as the density increases. However, at lower densities, its pruning is less effective, and additional operations such as calculating *EAT*

[1]https://transitfeeds.com
[2]https://www.diag.uniroma1.it//challenge9/download.shtml



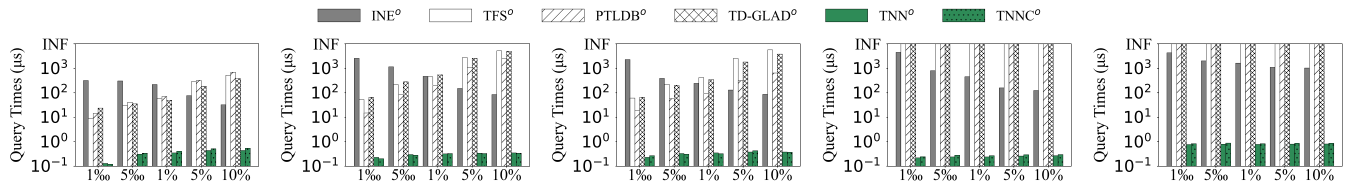
(a) SLC **(b) SYD** **(c) NO** **(d) UK** **(e) LKS**

Figure 4: Query time by varying density



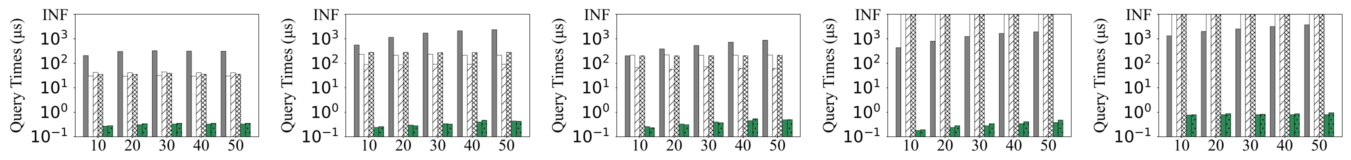
(a) SLC **(b) SYD** **(c) NO** **(d) UK** **(e) LKS**

Figure 5: Query time by varying k



(a) SLC **(b) SYD** **(c) NO** **(d) UK** **(e) LKS**

Figure 6: Query time by varying density (Opening hours)



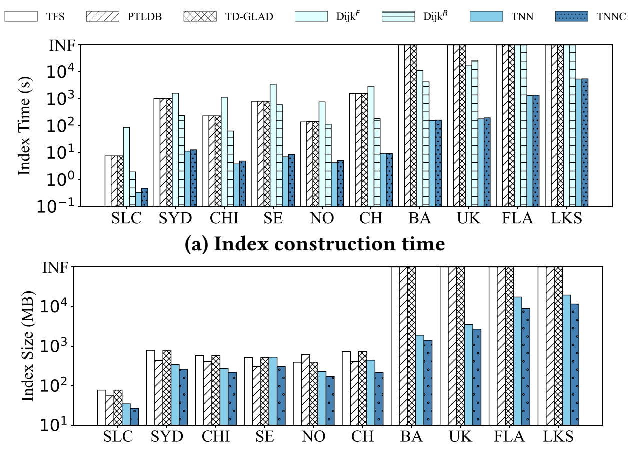
(a) SLC **(b) SYD** **(c) NO** **(d) UK** **(e) LKS**

Figure 7: Query time by varying k (Opening hours)

to boundary grids and maintaining the order of results lead to TD-GLAD taking more time than TFS. In contrast, TNN and TNNC algorithms not only show consistent query performance across varying densities but also maintain an outstanding query speed, they complete all the queries less than 1 microsecond in real-world networks and less than 2 microseconds in large synthetic networks, outperforming other algorithms by up to four orders of magnitude.

Exp-2: Query time by varying *k*. We also evaluate different methods by varying parameter *k*, their average query processing times are shown in Figure 5. As the value of *k* increases, the running time of INE also increases due to the need to find enough objects. TFS and PTLDB maintain a consistent running time regardless of k's value. TD-GLAD also exhibits an almost constant query time due to the ineffective pruning under the default object density. Our methods, TNN and TNNC, consistently outperform the other methods. There is a slight increase in running time as *k* increases, which can be attributed to the result size of the query.

Exp-3: Query time with opening hours. Figure 6 and Figure 7 presents the query processing times of different methods when considering opening hours with varying object densities and parameter *k*. Due to space limitations, we only present the results for five datasets. The results demonstrate that the query processing of the extension TNN^o is also significantly faster than the extension of other methods under different parameter configurations.



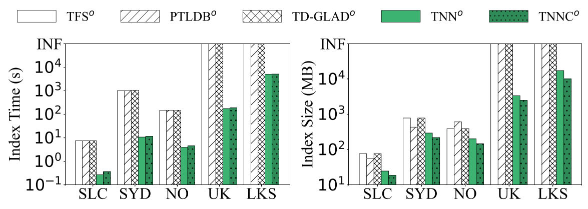
(a) Index construction time

(b) Index size

Figure 8: Indexing time and index size

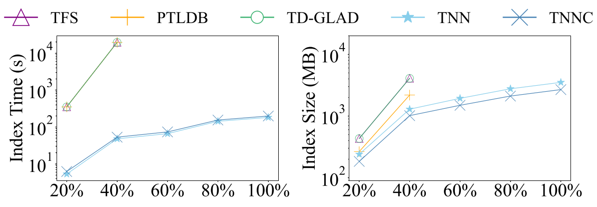
8.2 Index Construction

Exp-4: Indexing time and index size. The indexing time and index size of different methods are reported in Figure 8. Among these methods, the Dijk^F, Dijk^R and TNN are both used for constructing the TNN-Index. The results show that the index construction of TNN is significantly faster than other methods. Specifically, TNN



(a) Indexing time **(b) Index size**

Figure 9: Indexing time and index size (Opening hours)



(a) Indexing time **(b) Index size**

Figure 10: Scalability test (UK)

completes the index construction for the SE dataset in 7.08 seconds. In contrast, PTLDB costs 816.47 seconds and Dijk^R costs 605.88 seconds. This indicates a significant leap in processing speed. Moreover, TNNC adds a compression step that modestly increases the indexing time, a trade-off for its compression efficiency. Nonetheless, it still vastly outperforms existing methods in terms of indexing time. The advantage of TNNC becomes even more evident when considering index compression. In the SYD dataset, for example, TNNC achieves a compressed index size of 259.46 MB, compared to TFS's 782.19 MB, highlighting substantial gains in space efficiency. Compared to TFS, TD-GLAD only requires the additional step of building grids to store the objects. As a result, both methods have very similar indexing times and index sizes. Notably, the path index-based methods, TFS, PTLDB and TD-GLAD, fail to index the BA, UK, FLA and LKS datasets within the 8-hour time limit, demonstrating their inefficiency in handling large-scale datasets. In contrast, TNN methods successfully index all datasets promptly, demonstrating their robustness. Especially for the largest synthetic network LKS with 2.7 million vertices, TNNC can build an index with the size of 11.57 GB in 1.5 hours. Our proposed methods outperform existing methods with faster indexing times and smaller index sizes.

Exp-5: Indexing time and index size with opening hours. Figure 9 presents a comparison of the indexing times and the index sizes of the extensions of different methods that consider the opening hours. The index construction of TNN^o is significantly faster than TFS^o, PTLDB^o and TD-GLAD^o, achieving a speedup of more than 170 times, as evidenced in the CH dataset. Additionally, the sizes of the indexes generated by TNN^o are consistently smaller than those built by TFS^o, PTLDB^o and TD-GLAD^o.

Exp-6: Scalability test. We also conduct scalability tests for all the index-based methods using the largest real-world dataset UKBus. In the experiment, we generated four subgraphs of the UK dataset, each containing 20%, 40%, 60%, and 80% of the edges from the UK dataset. Figure 10 reports the indexing times and index sizes under different graph sizes. The experiment shows that the path index-based methods TFS, PTLDB and TD-GLAD failed to index the subgraph with more than 40% edges of the original graph within the 8-hour time limit, revealing their poor scalability on large datasets. In contrast, our method TNN shows excellent scalability as the graph size increases.

9 RELATED WORKS

***k*NN in road networks.** The kNN search problem in road networks can be solved using traversal-based methods [16, 24, 24, 26, 36]. Other methods [13, 21, 22] leverage the index to accelerate kNN queries. TOAIN [21] speeds up kNN queries in road networks with the Contraction-Hierarchy (CH) framework. GLAD [13] partitions a road network into grids and uses the H2H-Index [22] to speed up distance queries to the objects. TEN-QueryIP [23] is the state-of-the-art method, it pre-computes a kTNN index for each vertex, and processes queries with precomputed kTNN. These methods are designed for road networks with constant edge weights, they cannot be directly applied to PTNs.

***k*NN in time-dependent networks.** In time-dependent networks, travel times on edges are represented by piecewise linear functions. TD-NE [8] enhances INE [24] for kNN search in such networks, but its search complexity limits its efficiency. Komai et al. [15] introduced an index method dividing time into intervals with pre-computed minimum travel times. Demiryurek et al. [7] used a Voronoi diagram with TNI (Tight Network Index) and LNI (Loose Network Index) for efficient kNN queries.Yang et al. [33] developed a dynamic Voronoi-based index (V-tree) for more efficient querying. The state-of-the-art method for kNN queries in time-dependent networks is TD-GLAD [17]. It extends H2H-Index [22] for time-dependent networks. These methods are either inefficient in processing queries or rely on a path index, which is resource-intensive and their scalability is limited.

To address the kNN queries in public transportation networks, Li et al. [18] adapt Dijkstra's algorithm to process kNN queries in PTNs, but it is inefficient in processing queries. TFS [14] is designed for the top-k keyword search problem in PTNs, it uses a path index to compute the earliest arrival time for each object and returns the top-k objects with the earliest arrival time. [10] focused on SQL solutions within the PTLDB (Public Transportation Labeling on Databases) framework with TTL-Index as the underlying layer. It builds an inverse label for each hub vertex and uses it to speed up the query processing. However, these methods are inefficient in processing queries, and the underlying path index of the index-based methods is resource-intensive and cannot scale to large graphs.

10 CONCLUSIONS

In this paper, we present an efficient index, TNN-Index, for kNN searches in public transportation networks. We show that our approach can evaluate queries up to four orders of magnitude faster than existing methods. We present an efficient algorithm for constructing the index and a compression technique for minimizing the index. Furthermore, we extend the index to scenarios where the opening hours of objects are considered. Extensive experiments demonstrate the efficiency and effectiveness of our approach.

ACKNOWLEDGMENTS

Wenjie Zhang is supported by ARC DP230101445 and FT210100303. Lu Qin is supported by ARC FT200100787 and DP210101347. Ying Zhang is supported by ARC LP210301046, DP210101393 and DP230101445. Xuemin Lin is supported by NSFC U20B2046 and U2241211.

REFERENCES

[1] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. 1987. Complexity of finding embeddings in ak-tree. *SIAM Journal on Algebraic Discrete Methods* 8, 2 (1987), 277–284.

[2] Anne Berry, Pinar Heggernes, and Genevieve Simonet. 2003. The minimum degree heuristic and the minimal triangulation process. In *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 58–70.

[3] Hans L Bodlaender et al. 1992. A tourist guide through treewidth. (1992).

[4] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Hong Cheng, and Miao Qiao. 2012. The exact distance to destination in undirected world. *The VLDB Journal* 21 (2012), 869–888.

[5] Zitong Chen, Ada Wai-Chee Fu, Minhao Jiang, Eric Lo, and Pengfei Zhang. 2021. P2H: efficient distance querying on road networks by projected vertex separators. In *Proceedings of the 2021 International Conference on Management of Data*. 313–325.

[6] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[7] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. 2010. Efficient k-nearest neighbor search in time-dependent spatial networks. In *International Conference on Database and Expert Systems Applications*. Springer, 432–449.

[8] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. 2010. Towards k-nearest neighbor search in time-dependent spatial network databases. In *International workshop on databases in networked information systems*. Springer, 296–310.

[9] Edsger W Dijkstra. 2022. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*. 287–290.

[10] Alexandros Efentakis. 2016. Scalable Public Transportation Queries on the Database.. In *EDBT*. 527–538.

[11] Qingshuai Feng, You Peng, Wenjie Zhang, Ying Zhang, and Xuemin Lin. 2022. Towards real-time counting shortest cycles on dynamic graphs: A hub labeling approach. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 512–524.

[12] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. 2004. Distance labeling in graphs. *Journal of algorithms* 53, 1 (2004), 85–112.

[13] Dan He, Sibo Wang, Xiaofang Zhou, and Reynold Cheng. 2019. An efficient framework for correctness-aware kNN queries on road networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1298–1309.

[14] Wuwei Huang, Genan Dai, Youming Ge, and Yubao Liu. 2019. Top-k nearest keyword search in public transportation networks. In *2019 15th International Conference on Semantics, Knowledge and Grids (SKG)*. IEEE, 67–74.

[15] Yuka Komai, Duong Hong Nguyen, Takahiro Hara, and Shojiro Nishio. 2014. KNN search utilizing index of the minimum road travel time in time-dependent road networks. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*. IEEE, 131–137.

[16] 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.

[17] 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.

[18] Jiajia Li, Lingyun Zhang, Cancan Ni, Yunzhe An, Chuanyu Zong, and Anzhen Zhang. 2021. Efficient k Nearest Neighbor Query Processing on Public Transportation Network. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 1108–1115.

[19] Lei Li, Sibo Wang, and Xiaofang Zhou. 2020. Fastest path query answering using time-dependent hop-labeling in road network. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2020), 300–313.

[20] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2020. Scaling up distance labeling on graphs with core-periphery properties. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1367–1381.

[21] Siqiang Luo, Ben Kao, Guoliang Li, Jiafeng Hu, Reynold Cheng, and Yudian Zheng. 2018. Toain: a throughput optimizing adaptive index for answering dynamic k nn queries on road networks. *Proceedings of the VLDB Endowment* 11, 5 (2018), 594–606.

[22] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In *Proceedings of the 2018 International Conference on Management of Data*. 709–724.

[23] Dian Ouyang, Dong Wen, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Progressive top-k nearest neighbors search in large road networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1781–1795.

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

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

[26] Hanan Samet, Jagan Sankaranarayanan, and Houman Alborzi. 2008. Scalable network distance browsing in spatial databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 43–54.

[27] David Tedjopurnomo, Zhifeng Bao, Farhana Choudhury, Hui Luo, and AK Qin. 2022. Equitable Public Bus Network Optimization for Social Good: A Case Study of Singapore. In *2022 ACM Conference on Fairness, Accountability, and Transparency*. 278–288.

[28] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. 2015. Efficient route planning on public transportation networks: A labelling approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 967–982.

[29] Sheng Wang, Yuan Sun, Christopher Musco, and Zhifeng Bao. 2021. Public transport planning: When transit network connectivity meets commuting demand. In *Proceedings of the 2021 International Conference on Management of Data*. 1906–1919.

[30] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 99–110.

[31] Fang Wei-Kleiner. 2016. Tree decomposition-based indexing for efficient shortest path and nearest neighbors query answering on graphs. *J. Comput. System Sci.* 82, 1 (2016), 23–44.

[32] 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.

[33] Yajun Yang, Hanxiao Li, Junhu Wang, Qinghua Hu, Xin Wang, Muxi Leng, et al. 2019. A novel index method for k nearest object query over time-dependent road networks. *Complexity* 2019 (2019).

[34] Junhua Zhang, Wentao Li, Long Yuan, Lu Qin, Ying Zhang, and Lijun Chang. 2022. Shortest-path queries on complex networks: experiments, analyses, and improvement. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2640–2652.

[35] Junhua Zhang, Long Yuan, Wentao Li, Lu Qin, Ying Zhang, and Wenjie Zhang. 2024. Label-constrained shortest path query processing on road networks. *The VLDB Journal* 33, 3 (2024), 569–593.

[36] 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.