The VLDB Journal (2023) 32:575–594
https://doi.org/10.1007/s00778-022-00758-w

REGULAR PAPER

Efficient *k*NN query for moving objects on time-dependent road networks

Jiajia Li[1] · Cancan Ni[1] · Dan He[2] · Lei Li[3,4] · Xiufeng Xia[1] · Xiaofang Zhou[3,4]

Received: 16 October 2021 / Revised: 10 April 2022 / Accepted: 17 June 2022 / Published online: 27 July 2022

Abstract

In this paper, we study the Time-Dependent k Nearest Neighbor (TD-kNN) query on moving objects that aims to return k objects arriving at the query location with the least traveling cost departing at a given time t. Although the kNN query on moving objects has been widely studied in the scenario of the static road network, the TD-kNN query tends to be more complicated and challenging because under the time-dependent road network, the cost of each edge is measured by a cost function rather than a fixed distance value. To tackle such difficulty, we adopt the framework of GLAD and develop an advanced index structure to support efficient fastest travel cost query on time-dependent road network. In particular, we propose the Time-Dependent H2H (TD-H2H) index, which pre-computes the aggregated weight functions between each node to some specific nodes in the decomposition tree derived from the road network. Additionally, we establish a grid index on moving objects for candidate object retrieval and location update. To further accelerate the TD-kNN query, two pruning strategies are proposed in our solution. Apart from that, we extend our framework to tackle the time-dependent approachable kNN (TD-AkNN) query on moving objects targeting for the application of taxi-hailing service, where the moving object might have been occupied. Extensive experiments with different parameter settings on real-world road network show that our solutions for both TD-kNN and TD-AkNN queries are superior to the competitors in orders of magnitude.

Keywords k Nearest neighbor query · Time-dependent road network · Fastest travel time query

1 Introduction

With the proliferation of GPS-equipped devices, intelligent transportation services, e.g., DiDi [1], Uber [2], have served as essential travel tools for customers, which provide significant improvements against traditional service systems in terms of reducing taxi cruising time [3,4] and passengers' waiting time. Meanwhile, they also foster plenty of studies for location-based queries on road networks, among which the k nearest neighbor (kNN) query on moving objects plays an important role as a technical support for such service systems. In particular, given a query location, and a set of moving objects, the kNN query is to find the k nearest objects traveling from their current locations to the query location. The kNN query on moving objects has been widely studied [5–9] in the scenario of static road network, where the closeness between moving objects and the query location is measured by the road network distance.

In practice, however, the road networks are essentially *dynamic*, which means the traveling cost varies over time. On a time-dependent road network, the weight of each edge

✉ Dan He
d.he@uq.edu.au

Jiajia Li
lijiajia@sau.edu.cn

Cancan Ni
ni_cancan@163.com

Lei Li
thorli@ust.hk

Xiufeng Xia
xiaxiufeng@sau.edu.cn

Xiaofang Zhou
zxf@cse.ust.hk

1 Shenyang Aerospace University, Shenyang, China

2 The University of Queensland, Brisbane, Australia

3 Department of CSE, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong SAR, China

4 DSA Thrust, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China

is usually associated with a function of time, rather than a fixed distance value. Accordingly, the kNN query on such time-dependent road network aims to return k objects that could arrive fastest. In this case, the solutions for kNN query on static road network are no longer utilitarian. Motivated by this, in this paper, we study the *Time-Dependent k Nearest Neighbor* (*TD-kNN*) query on moving objects under time-dependent road networks, which to the best of our knowledge has not been resolved by any of the existing studies. In literature, time-dependent road network query has been extensively studied for years [10–21], while most of them focus on fastest travel path query of a given source and destination pair. Additionally, kNN query for static objects, e.g., point-of-interest, on time-dependent road network [22–26] has been well addressed, where travel cost from the query location to the static objects is concerned. Nevertheless, it is not straightforward to adopt the aforementioned two trends of algorithms to solve the TD-kNN query on moving objects due to the following challenges.

Challenges: In a time-dependent road network, the travel cost from the moving object to the query location is different from the one from the query location to object with the same departure time. For example, consider the real-world case that one travel from his/her home to the office in the morning. In common sense, the cost from home to office would be much larger compared to the one from office to home during the rush hour. Thus, the algorithms [24–26] that return the kNN objects departing from the query location would not be helpful. In the static road network, such issue could be easily handled by developing a reverse graph of the original graph, where the cost from the query location to the object in the reverse graph would be equivalent to the one from the object to the query location in the original graph. However, building a reverse graph for a time-dependent road network is challenging as the cost of each edge is measured by a function of time. Moreover, the queries on time-dependent road network are usually very time-consuming since the travel cost for each edge along a path depends on the cost for its previous edge. For instance, consider the travel cost for a path $\{v_1, v_2, v_5\}$ on the time-dependent graph in Fig. 1. Let $f_{(v_i,v_j)}(t)$ be the cost function of an edge (v_i, v_j). Then, the cost from v_2 to v_5 along this path is measured by $f_{(v_2,v_5)}(t + f_{(v_1,v_2)}(t))$. Consequently, without knowing the departure time, it is difficult to pre-compute the exact travel cost between some labeled vertices on the graph to accelerate the search.

Inspired from the framework of GLAD [6], which develops a light-weighted index to support efficient kNN query on moving objects under static road network, in this paper, we introduce an effective solution to solve TD-kNN query on moving objects. In particular, we use the uniform grid to index the moving objects, which is proved to be very efficient in updating the locations of moving objects. Then, we apply the expand-and-bound algorithm to explore the grid from the query location to retrieve the candidate objects. Next, the problem turns out to be how we can quickly refine the final results by computing the exact travel costs from the kNN objects to the query location. A naive way is to apply the existing algorithms for shortest path query on time-dependent road network, e.g., TD-Dijkstra [27], TD-G-tree [13] to calculate the travel cost for every candidate object. However, as we will show in our experiment, the computational cost for such solution is huge. To this end, in our paper, we develop a labeling index for time dependent travel cost calculation, named TD-H2H, which extends the H2H index [28] for static road network to adapt to the dynamic scenario. Such extension is not straightforward as the labeling information is not a distance value but an aggregated weight function, which will be discussed in Sect. 4. To further improve the query efficiency, we utilize a static H2H index on a lower-bound graph to prune some objects that would never be the kNN results, and propose several pruning strategies to avoid some unnecessary computation. In addition, we extend this framework to answer the time-dependent approachable kNN query considering the scenario that the moving objects contain two conditions (occupied/non-occupied), for applications like taxi-hailing service.

To sum up, our work has four primary contributions:

- We develop a TD-H2H index, which is able to return the fastest travel cost for any given pair of vertices on a time-dependent road network very efficiently.
- We introduce efficient and novel algorithm for time-dependent kNN query on moving objects, which has not been well addressed in literature. And, we propose several pruning strategies to further reduce the query time.
- We extend our proposed solution to tackle the time-dependent approachable kNN query on conditional moving objects considering more practical scenario.
- We conduct extensive experiments with real-world road network to show the superiority of our proposed solution against the competitors. And the results show that our algorithm is more efficient than the competitors in orders of magnitude.

2 Related work

In this section, we discuss the existing works of the kNN query on the static and time-dependent networks, and the moving object kNN on the static network.

2.1 Static *k*NN query on static road networks

This is the most fundamental type of kNN where the network distance never changes and the objects never move.

Firstly, ROAD [29] expands the network until k objects are found. It speeds up the expansion by skipping the non-moving objects sub-networks with the help of indexes and partitions. However, when the objects are distributed evenly, it would decay to the Dijkstra's algorithm. Therefore, the other approaches improve the efficiency by searching toward the promising objects with different heuristic functions and indexes [30]. For instance, TEN-QueryIP [31] pre-computes the top-k nearest objects for each vertex with the help of tree decomposition, and the query results are retrieved by combining these pre-computed top-k nearest objects. IER [32] computes the closest objects in Euclidean distance, until the minimum Euclidean distance is greater than the maximum network distance of the best k objects found. [30] uses PHL [33] as the underlying index for IER to speed up the retrieval of network shortest distance. Nevertheless, since the weights of the time-dependent road network vary over time, their indexes cannot solve the TD-kNN query directly. Moreover, because each vertex's top-k nearest objects are different at different time, and the moving object's position also keeps changing dynamically, the precomputed top-k information would become invalid and they can hardly be updated.

2.2 Static *k*NN query on time-dependent network

We first discuss the time-dependent routing briefly and then discuss the static kNN methods in this scenario.

2.2.1 Time-dependent route planning

The time-dependent networks use time-dependent functions, which is piecewise linear function, to describe the travel time of each edge on different departure time. When the departure time is a single time point, this problem is essentially the same as the static path problem and can be solved by any index-free shortest path algorithm. However, index cannot be built for it as it requires an index for each departure time. The more general time-dependent path problem deal with the cases when the departure time is an interval and finds the optimal departure time with the fastest path [10,11,17–19,34,35]. However, its complexity lowerbound is $\Omega(|S|(|V|log|V|+|E|))$ [36], where $|V|$ and $|E|$ are the vertex and edge number, and $|S|$ is the turning point number of the result's time-dependent function. It should be noted that $|S|$ is normally on the order of 2 or 3 magnitudes and no existing algorithm is close to this complexity, it would be very time-consuming to use them for TD-kNN query. Nevertheless, if we could build an index on it, it could answer both the interval departure and single departure query in constant time. However, the existing time-dependent indexes [13,15,16,20,37] are all search-based, which suffer from the high construction complexity (label / shortcut sizes $\times\Omega(|S|(|V|log|V|+|E|))$) and take a long time to construct. Besides, the time-dependent index size is also very big, which is normally their static versions' size times $|S|$, and it normally takes hundreds of GB in real life [15,20]. Therefore, the concatenation-based method proposed in this work aims to reduce the time-dependent index's high construction cost.

2.2.2 Static *k*NN on time-dependent network

Since the query objects are static, it is easy to extend the static methods to the time-dependent environment. TD-NE [22] extends the search-based methods with the single departure time fastest path. However, its performance is limited by the searching algorithm. More importantly, it cannot solve our problem because it retrieve k objects that can be reached the fastest from the query point, but not the objects that can reach the query point, and such operation is not supported since the time-dependent function cannot be reversed like the static value. [23] extends the index-based method by dividing the whole time interval into several sub-intervals and pre-computing the minimum travel time for each sub-interval. Then, during the search, an A*-like algorithm is utilized by referring to the minimum travel time from the index until kNNs are found. However, this index cannot facilitate queries effectively for large networks because the deviations are always too large between the estimated and the actual travel time. [25] utilizes the Voronoi diagram [24] with two complementary index structures: Tight Network Index (TNI) and Loose Network Index (LNI). The TNI/LNI cells for each data object are pre-computed based on the upper/lower bound of the travel time. If the query point locates in a TNI cell of an object, this object is definitely the nearest result of the query. Hence, the NN result can be obtained without performing any shortest path computation. On the contrary, if a query point is not in the TNI cell of any object, those objects whose LNI cell contains the query are taken as potential candidates, and they need to compute the shortest path for result verification. After that, [26] proposes another Voronoi-based index, which builds a Voronoi cell for object o when all the points within taking less time to o than to other objects at any departure time. Based on the Voronoi cells, the V-tree is constructed to manage all the closest cells to further speed up the query. However, both the Voronoi-based methods build the Voronoi cells based on the locations of objects. Once the object moves to a new location, multiple Voronoi cells will be affected and need to be re-calculated, which consumes huge computational costs in the moving objects scenarios.

2.3 Moving object *k*NN on static network

Different from the previous static kNN, the moving objects brings dynamic into this already complicated problem. Like the similar situation faced by the dynamic routing problem [38], the heavier (larger) index brings faster query perfor-

mance, but it requires longer time to cope with the update [39,40]. Therefore, the lighter index structure is preferable in the moving object scenario. TOAIN [5] builds a shortcut-based index called SCOB which a throughput optimizing adaptive index, and it stores at each *summit* node the precomputed kNN objects. The query time is significantly improved, while the update cost for the precomputed kNN objects is large to some extent. It proposes several settings to further get the trade-off between query and update cost so as to optimize the system throughput. GLAD [7] verifies the efficiency of this idea to solve the kNN query of moving objects. It utilizes the light-weighted grid index to manage the moving objects which make the location update cost very small. Then, the distance between the query point and the border point of the outermost expanded grids is taken as the lower bound of the shortest network distance, which is used to filter out the objects very far away from query and verify objects by calculating and comparing their distances to query. By integrating the H2H index [28], which is the state-of-the-art labeling-based solution for distances queries on the road network, GLAD significantly improves the throughput.

3 Preliminaries

In this section, we first introduce the important definitions, notations, and problem statement, and then, we describe our underlying structures H2H and GLAD.

3.1 Problem statement

Definition 1 *(Time-dependent directed graph)* A time-dependent road network is modeled as a directed graph $G = (V, E, F)$, where each vertex $v_i \in V$ represents road intersection, $(v_i, v_j) \in E$ is a directed edge from v_i to v_j, $f_{(v_i,v_j)} \in F$ is a time-dependent function of (v_i, v_j).

We use the *piecewise linear functions* [13] as the time-dependent function to describe the travel time of different departure time within the whole time domain $\mathbb{T} = [t_s, t_e]$. Specifically, given an edge (v_i, v_j), $f_{v_i,v_j}(t)$ is represented as a set of points $S = \{(t_1, f(t_1)), \ldots, (t_k, f(t_k))\}$, where $t_1 = t_s$ and $t_k = t_e$ as illustrated in Fig. 2.

Example 1 Figure 1 illustrates an example of a time-dependent graph with 9 vertices and 16 directed edges, where the interpolation points of piecewise linear weight function for each edge is given in Table 1. Consider the travel cost for edge (v_1, v_2) with a set of interpolation points $\{(0, 6), (20, 12), (40, 12), (60, 6)\}$. If a user departs at time 0 (resp.20), it takes 6 minutes (resp. 12 minutes) to travel from v_1 to v_2.

Definition 2 *(Time-Dependent Travel Cost (TTC))* Given a path $P = \langle v_1, v_2, \ldots, v_n \rangle$ from G and a departure time t,

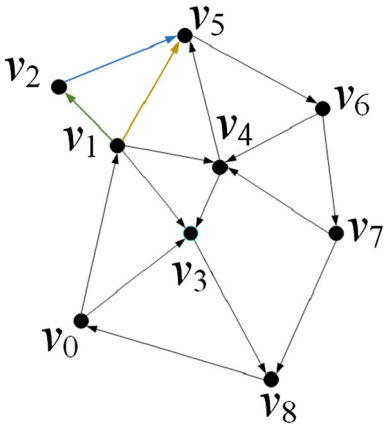


Fig. 1 Time-dependent directed graph

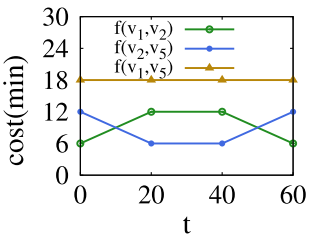


Fig. 2 Piecewise linear weight function for edge(v_1, v_5), (v_1, v_2), (v_2, v_5)

Table 1 Edge weights

Edge	Piecewise linear weight function
(v_0, v_1)	(0, 6), (20, 12), (60, 6)
(v_0, v_3)	(0, 3), (60, 3)
(v_1, v_2)	(0, 6), (20, 12), (40, 12), (60, 6)
(v_1, v_3)	(0, 12), (20, 12), (40, 6), (60, 12)
(v_1, v_4)	(0, 3), (40, 5), (60, 3)
(v_1, v_5)	(0, 18), (60, 18)
(v_2, v_5)	(0, 12), (20, 6), (40, 6), (60, 12)
(v_3, v_8)	(0, 24), (60, 24)
(v_4, v_3)	(0, 3), (60, 3)
(v_4, v_5)	(0, 20), (60, 20)
(v_5, v_6)	(0, 12), (20, 24), (60, 12)
(v_6, v_4)	(0, 2), (60, 2)
(v_6, v_7)	(0, 6), (20, 12), (40, 6), (60, 6)
(v_7, v_4)	(0, 8), (20, 5), (40, 8), (60, 8)
(v_7, v_8)	(0, 20), (60, 20)
(v_8, v_0)	(0, 24), (20, 12), (40, 12), (60, 24)

the time-dependent travel cost of this path is calculated by the following equation:

$$TTC(P, t) = \sum_{i=1}^{n-1} f_{(v_i, v_{i+1})}(t_i),$$

where $t_1 = t, t_i = t_{i-1} + f_{(v_{i-1},v_i)}(t_{i-1}), i = 2, \ldots, n-1$.

Definition 3 *(Time-Dependent Fastest Travel Cost (TFTC))* Given a source s, a destination d in G, and a departure time t, let $\mathbb{P}$ be the set of all possible paths from s to d in G. We have the time-dependent fastest travel cost from s to d defined as:

$$TFTC(s, d, t) = \min_{P \in \mathbb{P}} TTC(P, t)$$

Example 2 Consider the example in Fig. 1. Given a path $P = \langle v_1, v_2, v_5 \rangle$ with a departure time 20, we have $TTC(P, 20) = 12 + 6 = 18$. Given a source and destination pair v_1 and v_5, and the departure time 20, there are three possible paths: $P_1 = \langle v_1, v_5 \rangle$, $P_2 = \langle v_1, v_2, v_5 \rangle$, $P_3 = \langle v_1, v_4, v_5 \rangle$. Correspondingly, $TTC(P_1, 20) = 18$, $TTC(P_2, 20) = 18$, and $TTC(P_3, 20) = 24$. Thus, $TFTC(v_1, v_5, 20) = 18$.

Definition 4 *(Lower Bound Graph)* Given a time-dependent graph $G(V, E, F)$, the corresponding lower bound graph $G_l = (V_l, E_l)$ is a static directed graph, where $V = V_l$, $E = E_l$, and the weight for an edge $(v_i, v_j) \in E_l$ is static and equal to the minimum travel cost from the time-dependent weight function $f_{(v_i,v_j)}$.

Definition 5 *(Lower Bound Travel Cost)* Given a path $P_l = \langle v_1, v_2, \ldots, v_n \rangle$ from G_l, the travel cost for P_l, denoted by $C_{G_l}(P_l)$ is the sum of the weights along the path. Given a source vertex s and a destination vertex d in G_l, let $\mathbb{P}_{\ll}$ be all the possible path from s to d. The lower bound travel cost from s to d is defined as $LBC(s, d) = \min_{P \in \mathbb{P}_{\ll}} C_{G_l}(P)$.

With the *lower bound graph*, we are able to determine the minimum travel cost for any given source and destination pair, regardless of the departure time. In this paper, we assume that G suffices the First-In-First-Out (FIFO) property, and the moving objects do not wait at any vertex while traversing along a path. Consider a set M of moving objects on the time-dependent road network. Following previous works [5,7], we assume that all moving objects are located on vertices. Given a query point q, a moving object $o \in M$ on a time-dependent road network, to travel from o to q departing at time t, the minimum travel cost is calculated by $TFTC(o, q, t)$. Accordingly, in our work, we study the k nearest neighbor (kNN) query on time-dependent road network, which is defined as follows.

Definition 6 *(Time-Dependent kNN (TD-kNN))* Given a query point q, a set M of moving objects on a time-dependent road network $G(V, E, F)$, a departure time t, and an integer $k \leq |M|$, time-dependent kNN returns a subset $R \subseteq M$ of k moving objects such that $\forall o_i \in M \setminus R, TFTC(o_i, q, t) \geq TFTC(o_j, q, t), \forall o_j \in R$.

3.2 Background knowledge

In this section, we introduce some background knowledge of H2H index and the framework of GLAD, which are the foundation of our proposed solution.

3.2.1 H2H index

H2H [28] index uses tree decomposition to organize the graph cut information and distance labels to support efficient distance query on a graph. Specifically, the tree decomposition maps a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (regardless of the edge weight) to a tree as follows:

Definition 7 *(Tree Decomposition* [41]*)* A decomposition tree of a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, denoted as $T_{\mathcal{G}}$, is a rooted tree in which each node $X \in V(T_{\mathcal{G}})$ is a subset of $\mathcal{V}$ (i.e., $X \subseteq \mathcal{V}$) such that the following three conditions hold:

(1) $\cup_{X \in \mathcal{V}} X = \mathcal{V}$;

(2) $\forall (u, v) \in \mathcal{E}, \exists X \subset \mathcal{V}$ s.t. $u \in X$ and $v \in X$;

(3) $\forall v \in \mathcal{V}$ the set $\{X \mid v \in X\}$ forms a connected subtree of $T_{\mathcal{G}}$.

For the rest of paper, we use $vertex$ for a vertex in the road network graph, and $node$ for a node in the $T_{\mathcal{G}}$. We notate $T(v)$ as the subtree induced by the set $\{X | v \in X\}$ in $T_{\mathcal{G}}$ and $X(v)$ as the root node of $T(v)$. The largest size of $X(v)$ is denoted as tree-width W, and the tree height is denoted as h.

Definition 8 *(Vertex Cut* [28]*)* Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, a subset of vertices $\mathcal{V}_c \subset \mathcal{V}$ is a vertex cut of $\mathcal{G}$ if the deletion of $\mathcal{V}_c$ from $\mathcal{G}$ splits $\mathcal{G}$ into multiple connected components. A vertex set $\mathcal{V}_c$ is called the vertex cut of vertices u and v if u and v are in different connected components by the deletion of $\mathcal{V}_c$ from $\mathcal{G}$.

Property 1 Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, its decomposition tree $T_{\mathcal{G}}$, and an arbitrary node $X(u) \in V(T_{\mathcal{G}})$, for any $v \in X(u) \backslash \{u\}$, $X(v)$ is an ancestor of $X(u)$ in $T_{\mathcal{G}}$.

Property 2 Given a decomposition tree $T_{\mathcal{G}}$ for a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, for any two vertices u and v in $\mathcal{V}$, suppose $X(u)$ is not an ancestor/decedent of $X(v)$ in $T_{\mathcal{G}}$, let X be the lowest common ancestor(LCA) of $X(u)$ and $X(v)$ in $T_{\mathcal{G}}$, then X is a vertex cut of u and v of $\mathcal{G}$.

Based on the above properties, we have that the shortest path between two vertices must pass through a vertex in a vertex cut set. And the LCA of two vertices in the decomposition tree is a vertex cut. Hence, the following theorem can be easily obtained.

Theorem 1 *Given a road network $\mathcal{G}$, let C be a vertex cut for two vertices v and u, we have:*

$$dist(v, u) = \min_{w \in C} dist(v, w) + dist(w, u) \quad (1)$$

To speed up the distance query efficiency, H2H precomputes the shortest distances between vertex to its ancestor and stores them in multiple arrays which are defined as follows.

Definition 9 *(Ancestor Array* [28]*)* Given a decomposition tree $T_{\mathcal{G}}$ for a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, for any node $X(v) \in V(T_{\mathcal{G}})$, let $\{X(w_1), X(w_2), \ldots, X(w_l)\}$ be the path from the root of $T_{\mathcal{G}}$ to $X(v)$ in $T_{\mathcal{G}}$, where $X(w_1)$ is the root node and $X(w_l) = X(v)$. The ancestor array of $X(v)$, denoted by $X(v).anc$ is defined as $X(v).anc = \{w_1, w_2, \ldots, w_l\}$. Here, we denote $X(v).anc_i$ as the i-th element in $X(v).anc$ for any $1 \leq i \leq l$.

Distance Array. The distance array for $X(v)$ is defined as $X(v).dis = \{dist(v, w_1), dist(v, w_2), \ldots, dist(v, w_l)\}$, which indicates of distances from v to every vertex in $X(v).anc$.

Position Array. The *position array* for $X(v)$ is defined as $X(v).pos = \{p_1, p_2, \ldots, p_l\}$, where p_i $(1 \leq i \leq l)$ is the position of w_i in $X(v).anc$, i.e., $X(v).anc_{p_i} = w_i$. We sort the values in $X(v).pos$ in ascending order. And we denote $X(v).pos_i$ as the i-th value in $X(v).pos$ for any $1 \leq i \leq l$.

Given the above distance and position arrays, the distance query for any two vertices can be computed as follows. We first calculate the LCA of these two vertices in the decomposition tree and then scan the position and distance arrays to obtain the minimum accumulative distance from these vertices to the LCA, which is the shortest distance between the corresponding vertices.

3.2.2 GLAD

GLAD is a grid-based query framework for kNN query on moving objects in static road network. The basic idea is that GLAD takes H2H index as a black-box for distance query, and it partitions the road network into uniform grids to record the moving objects. When using GLAD index to answer kNN query, we start from where the query point is located to explore the grid to obtain the candidate objects, and calculate the distance from each object to the query point by H2H index. We narrow down the exploration space when more candidate objects are obtained until the minimum distance from the next object to explore exceeds the largest distance from the candidate objects to the query point. Finally, we refine the kNN results from the retrieved candidate objects.

4 TD-kNN query processing

Compared to the kNN query on the static road network, the major challenge to answer TD-kNN query is that the travel cost for any source and destination pair of vertices varies over time, and we can only determine the corresponding cost when a query is issued. Thus, it is impossible to pre-store some intermediate kNN results for a certain vertex to accelerate the query processing, like it does in many existing works, e.g., V-tree [8], TOAIN [5]. Moreover, though fastest path queries on time-dependent road network have been well studied [12], it is not straightforward to directly apply the existing solutions, e.g., TD-Dijkstra [27], TD-G-tree [13], to address TD-kNN query.

In particular, consider the simple TD-Dijkstra algorithm on the fastest path query. When given a pair of source and destination vertices s and d on a time-dependent road network G, the TD-Dijkstra algorithm starts from s to explore G and accumulates the cost based on the weight functions at the current time visiting a vertex. The algorithm terminates until it reaches d and returns the total cost from s to d. Note that, unlike the distance-based shortest path query, even in an undirected graph, the TD-Dijkstra returns different costs for the paths from s to d and from d to s, since the cost varies with the time reaching a vertex. On the other hand, applying Dijkstra's algorithm for the kNN query on the static road network, one can simply start from the query point and explore the road network with a reverse topological structure until obtaining the top-k objects encountered during the exploration. However, in terms of the TD-kNN query, a similar strategy on TD-Dijkstra is incapable of achieving the correct results. This is because starting from the query point to explore the graph can only return the nearest objects departing from the query point, while based on our definition, we aim to obtain the objects that can arrive at the query point with the least travel cost and the travel cost from a query vertex to an object is completely different with the one from the object to the query under the dynamic weight circumstance. A similar issue also appears in TD-G-tree.

To tackle the aforementioned challenges, in this paper, we extend the framework of GLAD [6] to handle TD-kNN query, which develops a grid and H2H [28] labeling index for kNN query on moving objects under static road network. Recall that in GLAD, the road network is partitioned into uniformed grids, and then, a data structure is maintained to record the moving objects that fall into the corresponding girds. In addition, a distance labeling scheme H2H [28] on road network is integrated for efficient distance query between two vertices. To answer the kNN query, we start from the grid where the query point locates and expand the grid from size 1×1 to $3 \times 3, 5 \times 5, \ldots$ until the kNN objects are retrieved. Whenever an object is found, we invoke the distance scheme H2H to calculate the exact road network distance between the moving object and the query location so as to refine the final kNN results. The remarkable advantage of this framework is that the update cost of the index for moving objects is slight, even when the locations of objects are updated very frequently. Refer to the original paper [6] for more details.

In our work, we adopt the same paradigm as GLAD for the TD-kNN query, where we first search for the candidate objects, followed by travel cost calculation for refinement of final results. However, the current index structure from GLAD is considerably insufficient for the time-dependent scenario since the H2H index can only answer static distance queries between two vertices. Alternatively, we can replace the phase of distance query with the time-dependent fastest travel cost query by applying the TD-Dijkstra algorithm or the TD-G-tree index. Nevertheless, the computational cost would be huge, since the time complexity for each time-dependent fastest travel cost by applying TD-Dijkstra is at $O(|V| \cdot log|V| + |E|)$ and that by TD-G-tree costs $O(log_2{}^2 k_f \cdot |V| \cdot log_2{}^2 |S|)$ [13]. It should be noted that $|S|$ here is the interpolation point number in the labels' functions so it is much larger than the edge functions'. In this paper, to improve the efficiency of the travel cost query, we develop a *time-dependent H2H (TD-H2H)* index by improving the original H2H index to support the time-dependent fastest travel cost query, which exactly makes up for the deficiency of the framework in GLAD.

4.1 TD-H2H index structure

In the rest of this paper, we consider a decomposition tree T_G on time-dependent directed graph G. We first give a theorem derived from the definition of vertex cut.

Theorem 2 *Given a time-dependent road network G, let C be a vertex cut for two vertices v and u, we have:*

$$TFTC(v,u,t) = \min_{w \in C} TFTC(v,w,t) + TFTC(w,u,t + TFTC(v,w,t)) \quad (2)$$

Next, we introduce the TD-H2H index, which extends the H2H to build the labeling index based on time-dependent road network. Given two adjacent edges, (v_i, v_j), (v_j, v_k), and their corresponding weight functions $f_{(v_i,v_j)}$, $f_{(v_j,v_k)}$, we can aggregate their weight functions to derive the weight function for the path $\langle v_i, v_j, v_k \rangle$, denoted by $f_{(v_i \rightsquigarrow v_k)}$, as follows:

$$\begin{aligned} f_{(v_i \rightsquigarrow v_k)}(t) =& (f_{(v_i,v_j)} \oplus f_{(v_j,v_k)})(t) \\ =& f_{(v_i,v_j)}(t) + f_{(v_j,v_k)}(t + f_{(v_i,v_j)}(t)) \end{aligned}$$

Then, we formally define the aggregated weight function for given two paths from v_i to v_j and from v_j to v_k as follows. In the rest of this paper, we use both $f_{(v,u)}$ and $f_{(v \rightsquigarrow u)}$ to denote the weight function for an edge (v, u).

Definition 10 *(Aggregated Weight Function)* Given two paths, $\langle v_i, \ldots, v_j \rangle$, $\langle v_j, \ldots, v_k \rangle$, and their corresponding weight functions $f_{(v_i \rightsquigarrow v_j)}$, $f_{(v_j \rightsquigarrow v_k)}$, we define the aggregated weight functions for the concatenated path $\langle v_i, \ldots, v_j, \ldots, v_k \rangle$, denoted by $f_{(v_i \rightsquigarrow v_k)}$ as follows.

$$\begin{aligned} f_{(v_i \rightsquigarrow v_k)}(t) =& f(f_{(v_i \rightsquigarrow v_j)} \oplus f_{(v_j \rightsquigarrow v_k)})(t) \\ =& f_{(v_i \rightsquigarrow v_j)}(t) + f_{(v_j \rightsquigarrow v_k)}(t + f_{(v_i \rightsquigarrow v_j)}(t)) \end{aligned}$$

Min Operation on Weight Functions. Given two different weight functions regarding the paths with the same source v_i and destination v_j $\langle v_i, \ldots, v_j \rangle$, denoted by $f'_{(v_i \rightsquigarrow v_j)}$ and $f''_{(v_i \rightsquigarrow v_j)}$, we define the min operation as follows.

$$\text{Min}\{f'_{(v_i \rightsquigarrow v_j)}, f''_{(v_i \rightsquigarrow v_j)}\} = \min_{\forall t \in [t_s, t_e]} \{f'_{(v_i \rightsquigarrow v_j)}(t), f''_{(v_i \rightsquigarrow v_j)}(t)\}$$

Note that after the Min operation, the new weight function for a pair of source and destination, e.g., $(v_i \rightsquigarrow v_j)$, might refer to multiple paths from the source to the destination. That is, at any time, it always chooses the path with the least traveling cost.

Example 3 Given two adjacent edges, (v_1, v_2), (v_2, v_5) in Fig. 1, with weight functions $f_{(v_1,v_2)}$ = {(0, 6), (20, 12), (40, 12), (60, 6)}, $f_{(v_2,v_5)}$ = {(0, 12), (20, 6), (40, 6), (60, 12)}, we have $f_{(v_1,v_2)} \oplus f_{(v_2,v_5)}$= {(0, 17), (11, 15), (20, 18), (28, 18), (40, 21), (60, 21)}. Since $f_{(v_1,v_5)}$ = {(0, 18), (60, 18)}, we have Min$\{f_{(v_1,v_5)}, f_{(v_1,v_2)} \oplus f_{(v_2,v_5)}\}$ = {(0, 17), (11, 15), (20, 18), (60, 18)}. Moreover, from 0 to 20 ($v_1 \rightsquigarrow v_5$) takes path $\{v_1, v_2, v_5\}$, and during 20 to 60, ($v_1 \rightsquigarrow v_5$) indicates the edge (v_1, v_5).

Based on the above definitions, we introduce the vertex elimination that preserve the weight functions as follows.

Definition 11 Function Preserved Vertex Elimination. Given a graph $G(V, E, F)$ and a vertex $v \in V$, the Vertex Elimination operation on v in G is as follows: For every pair of neighbors (u, w) of v, if $(u, v) \in E$, $(v, w) \in E$ and $(u, w) \notin E$, a new edge (u, w) with weight $f_{(u \rightsquigarrow w)}$ is inserted. Otherwise, if $(u, w) \in E$, we update the weight function for (u, w) in the whole time interval $\mathbb{T}$ by $f_{(u \rightsquigarrow w)}$ = Min$\{f_{(u,w)}, f_{(u,v)} \oplus f_{(v,w)}\}$. Then, v is removed.

Algorithm 1 gives the pseudocode of function preserved vertex elimination for one vertex on a given graph. Accordingly, we introduce the algorithm of function preserved tree decomposition, which is similar with the tree decomposition algorithm in [28]. The pseudocode is shown in Algorithm 2. In particular, let H be a duplication of G and $N(v, H)$ indicates the neighbors of v in H, which includes all in-neighbors and out-neighbors. Lines 3-8 iteratively eliminates the vertex v with the smallest degree in graph H. And for each v, we create a node $X(v)$ in T_G, which is a star containing not only the vertices $\{v\} \cup N(v, H)$, but also the edges (v, u) with

Algorithm 1: Function Preserved Vertex Elimination (FPVE)

Input: graph $G(V, E, F)$, a vertex $v \in V$
Output: the function preserved graph H
1 Let $N(v)$ be the neighbors of v in G;
2 $H \leftarrow G$;
3 **for** *all pair of vertices* $u, w \in N(v)$ **do**
4 **if** $(u, v) \in E$ *and* $(v, w) \in E$ **then**
5 **if** $(u, w) \notin E$ **then**
6 insert edge (u, w) with $f_{(u \rightsquigarrow w)} = f_{(u,v)} \oplus f_{(v,w)}$;
7 **else**
8 $f_{(u \rightsquigarrow w)} \leftarrow \text{Min}\{f_{(u,w)}, f_{(u,v)} \oplus f_{(v,w)}\}$;
9 Return H;

Algorithm 2: Function Preserved Tree Decomposition (FPTD)

Input: A time dependent road network $G(V, E, F)$
Output: Functions Preserved Tree Decomposition T_G
1 $H \leftarrow G$; $T_G \leftarrow \emptyset$;
2 Let $N(v, H)$ be the neighbors of v in H;
3 **for** $i = 1$ *to* $|V|$ **do**
4 $v \leftarrow$ the vertex in H with smallest degree;
5 $X(v) \leftarrow$ the star from v to each neighbor $u \in N(v, H)$;
6 Create a node $X(v)$ in T_G;
7 $H \leftarrow \text{FPVE}(H, v)$;
8 $\pi(v) \leftarrow i$;
9 **for** *all* $v \in V$ **do**
10 **if** $|X(v)| > 1$ **then**
11 $u \leftarrow$ the vertex in $X(v) \setminus \{v\}$ with smallest π value;
12 Set the parent of $X(v)$ be $X(u)$ in T_G;
13 **for** *all* $v \in V$ **do**
14 Sort vertices in $X(v)$ in decreasing order of π values;
15 $X(v).\phi_i^o \leftarrow f_{(v,x_i)}$ where x_i is the i-th vertex in $X(v)$ for all $1 \le i \le |X(v)|$;
16 $X(v).\phi_i^i \leftarrow f_{(x_i,v)}$ where x_i is the i-th vertex in $X(v)$ for all $1 \le i \le |X(v)|$;
17 Return T_G;

weight function $f_{(v,u)}$ for all $u \in N(v, H)$. We construct the tree structure by assigning the parent node for each $X(v)$ in Lines 9-12. For the sake of efficiency, we sort the vertices in $X(v)$ in decreasing order of π value, which is assigned to each vertex in Line 8. Then, in Lines 13-16 we maintain a data structure $X(v).\phi_i^i$ (resp. $X(v).\phi_i^o$) for the weight functions of the edges from the in-neighbor x_i to v (resp. from v to the out-neighbor x_i). Here, we suppose $f_{(v,v)} = (0, 0), (60, 0)$ $(t \in [0, 60])$. Finally, we return T_G as the decomposition tree in Line 17.

Example 4 Consider the example in Fig. 1. To construct T_G, we first pick v_2, and create a node $X(v_2)$ in T_G which contains a star with two edges (v_1, v_2), (v_2, v_5) and the corresponding weight functions as shown in Table 1. We eliminate v_2 by updating edge (v_1, v_5) with $f_{(v_1 \rightsquigarrow v_5)} = \text{Min}\{f_{(v_1,v_5)}, f_{(v_1,v_2)} \oplus f_{(v_2,v_5)}\} = \{(0, 17), (11, 15), (20, 18), (60, 18)\}$. The process stops when all vertices are eliminated. For node $X(v_2)$, after sorting the vertices in $X(v_2)$ in decreasing order of their π values, we obtain the order v_1, v_5, v_2. Thus, we have $X(v_2).\phi^o = \{f_{(v_2,v_5)}, f_{(v_2,v_2)}\}$, $X(v_2).\phi^i = \{f_{(v_1,v_2)}, f_{(v_2,v_2)}\}$.

Similar to the H2H index that uses arrays to pre-store distances, in our TD-H2H we maintain the function arrays for each node in the decomposition tree T_G, which is defined as follows.

Function Arrays. The *function arrays* for $X(v)$ is defined as $X(v)^o.f = (f_{(v \rightsquigarrow w_1)}, f_{(v \rightsquigarrow w_2)}, \ldots, f_{(v \rightsquigarrow w_l)})$ and $X(v)^i.f = (f_{(w_1 \rightsquigarrow v)}, f_{(w_2 \rightsquigarrow v)}, \ldots, f_{(w_l \rightsquigarrow v)})$. To explain, the function array of $X(v)$ is the array for the weight functions from v to every vertex in $X(v).anc$ and from every vertex in $X(v).anc$ to v. We use $X(v)^o.f_i$ to denote the i-th value in $X(v)^o.f$ and $X(v)^i.f_i$ to denote the i-th value in $X(v)^i.f$ for any $1 \le i \le l$. We have $X(v)^o.f_i = f_{(v \rightsquigarrow X(v).anc_i)}$ and $X(v)^i.f_i = f_{(X(v).anc_i \rightsquigarrow v)}$.

In addition, for efficient lookup of weight functions in the function arrays, we also maintain the position arrays for each node in the decomposition tree as follows.

Example 5 For the node $X(v_8) = \{v_1, v_5, v_3, v_8\}$ with $X(v_8).anc = \{v_4, v_1, v_5, v_3, v_8\}$ in Fig. 3, we have the function arrays as $X(v_8)^o.f = (f_{(v_8 \rightsquigarrow v_4)}, f_{(v_8 \rightsquigarrow v_1)}, f_{(v_8 \rightsquigarrow v_5)}, f_{(v_8 \rightsquigarrow v_3)}, f_{(v_8 \rightsquigarrow v_8)})$ and $X(v_8)^i.f = (f_{(v_4 \rightsquigarrow v_8)}, f_{(v_1 \rightsquigarrow v_8)}, f_{(v_5 \rightsquigarrow v_8)}, f_{(v_3 \rightsquigarrow v_8)}, f_{(v_8 \rightsquigarrow v_8)})$. And the position array for $X(v_8).pos = \{2, 3, 4, 5\}$ since v_1, v_5, v_3, and v_8 are the $2nd$, $3rd$, $4th$ and $5th$ value in $X(v_8).anc$.

Next, we introduce the algorithm to construct the TD-H2H index. The main idea to build the TD-H2H index is to

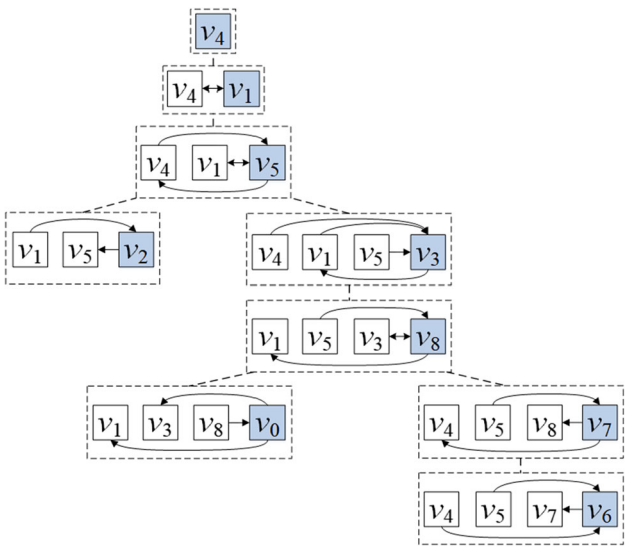


Fig. 3 Tree decomposition T_G

first construct the function preserved tree decomposition T_G and then generate the aforementioned function arrays and position array for each tree node.

Let $G(v)$ be the union of all the stars for nodes in the path from $X(v)$ to the root of T_G. We have the graph $G(v)$ to be a supergraph of $G(u)$ for any $u \in X(v).anc$. In this case, we can reuse the partial label information in $X(u)$ while computing the label for $X(v)$. Hence, we calculate the vertex labels in a top-down manner in T_G. In particular, for any $x_j \in X(v)\backslash\{v\}$, which is a neighbor of v, we have $X(x_j)$ is an ancestor of $X(v)$ in T_G based on Property 1. Thus, before we calculate the function arrays of v, the weight function $f_{(x_j \rightsquigarrow X(v).anc_i)}$ and $f_{(X(v).anc_i \rightsquigarrow x_j)}$ have been obtained. Moreover, for any $x_j \in X(v)$, we have $f_{(v \rightsquigarrow x_j)} = X(v).\phi^o_j$ and $f_{(x_j \rightsquigarrow v)} = X(v).\phi^i_j$ by Algorithm 2. Thus, we can calculate the function arrays for every vertex with the following lemma.

Lemma 1 *For any $1 \le i < |X(v).anc|$, we have*

$$X(v)^o.f_i = Min\{X(v)^o.f_i, X(v).\phi^o_j \oplus f_{(x_j \rightsquigarrow X(v).anc_i)}\}$$
$$X(v)^i.f_i = Min\{X(v)^i.f_i, f_{(X(v).anc_i \rightsquigarrow x_j)} \oplus X(v).\phi^i_j\}$$

$\forall j \in [1, |X(v)|)$, *where x_j is the j-th vertex in $X(v)$.*

According to Lemma 1, to obtain the function arrays, we need to compute $f_{(x_j \rightsquigarrow X(v).anc_i)}$ and $f_{(X(v).anc_i \rightsquigarrow x_j)}$ for each $x_j \in X(v)$ $(1 \le i < |X(v).anc|)$. We consider the following two case:

- **Case1:** $X(v).pos_j > i$. In this case, $X(X(v).anc_i)$ is an ancestor of $X(x_j)$ in T_G. Thus, we have $f_{(x_j \rightsquigarrow X(v).anc_i)} = f_{(x_j \rightsquigarrow X(x_j).anc_i)} = X(x_j)^o.f_i$ and $f_{(X(v).anc_i \rightsquigarrow x_j)} = f_{(X(x_j).anc_i \rightsquigarrow x_j)} = X(x_j)^i.f_i$. Moreover, $X(x_j)^o.f_i$ and $X(x_j)^i.f_i$ can be directly used as $f_{(x_j \rightsquigarrow X(v).anc_i)}$ and $f_{(X(v).anc_i \rightsquigarrow x_j)}$ because of the top-down traversal.
- **Case2:** $X(v).pos_j \le i$. In this case, $X(x_j)$ is an ancestor of $X(X(v).anc_i)$ in T_G. Then, we have $f_{(x_j \rightsquigarrow X(v).anc_i)} = X(X(v).anc_i)^i.f_{X(v).pos_j}$ and $f_{(X(v).anc_i \rightsquigarrow x_j)} = X(X(v).anc_i)^o.f_{X(v).pos_j}$. Similar with Case 1, $X(X(v).anc_i)^o.f_{X(v).pos_j}$ and $X(X(v).anc_i)^i.f_{X(v).pos_j}$ can be directly used as $f_{(x_j \rightsquigarrow X(v).anc_i)}$ and $f_{(X(v).anc_i \rightsquigarrow x_j)}$, respectively.

The pseudocode of the algorithm to construct TD-H2H index is shown in Algorithm 3. First, we initialize the maximum and minimum weight functions f_{max} and f_{min} in Line 1. In Line 2, we construct the decomposition tree T_G invoking the Algorithm 2. Lines 3–20 iteratively examine all nodes $X(v)$ in T_G in a top-down manner. In particular, for each node $X(v)$, we first compute its position array $X(v).pos$ in Lines 5–6. Then, in the descending order of their position, Lines 7–18 compute the function arrays $X(v)^o.f_i$ (resp. $X(v)^i.f_i$) for the weight functions from v to the i-th ancestor, $X(v).anc_i$ (resp. from $X(v).anc_i$ to v). After the initialization in Lines 7–9, for each element x_j (except for the last one) in $X(v)$, we first compute $f_{(x_j \rightsquigarrow X(v).anc_i)}$ and $f_{(X(v).anc_i \rightsquigarrow x_j)}$ based on Lemma 1 and the aforementioned two cases. Then, we conduct a Min operation on $X(v)^o.f_i$ and $X(v).\phi^o_j \oplus f_{(x_j \rightsquigarrow X(v).anc_i)}$ according to Definition 10 in Lines 17–18. In Lines 19–20, we set the last element in the function array to be f_{min} since they are the weight functions between v and itself. Finally, we return the function and position arrays for all nodes as the TD-H2H index in Line 21, which finish the TD-H2H index construction.

Algorithm 3: TD-H2H Index

Input: The time-dependent road network $G(V, E, F)$
Output: The TD-H2H-Index
1 **Initialize** $f_{max} = \{(0, +\infty), (60, +\infty)\}$, $f_{min} = \{(0, 0), (60, 0)\}$;
2 $T_G \leftarrow$ FPTD(G);
3 **for** *all* $X(v) \in V(T_G)$ *in a top-down manner* **do**
4 Suppose $X(v) = (x_1, x_2, ..., x_{|X(v)|})$;
5 **for** $i = 1$ *to* $|X(v)|$ **do**
6 $X(v).pos_i \leftarrow$ the position of x_i in $X(v).anc$;
7 **for** $i = 1$ *to* $|X(v).anc| - 1$ **do**
8 $X(v)^o.f_i \leftarrow f_{max}$;
9 $X(v)^i.f_i \leftarrow f_{max}$;
10 **for** $j = 1$ *to* $|X(v)| - 1$ **do**
11 **if** $X(v).pos_j > i$ **then**
12 $f_{(x_j \rightsquigarrow X(v).anc_i)} = X(x_j)^o.f_i$;
13 $f_{(X(v).anc_i \rightsquigarrow x_j)} = X(x_j)^i.f_i$;
14 **else**
15 $f_{(x_j \rightsquigarrow X(v).anc_i)} = X(X(v).anc_i)^i.f_{X(v).pos_j}$;
16 $f_{(X(v).anc_i \rightsquigarrow x_j)} = X(X(v).anc_i)^o.f_{X(v).pos_j}$;
17 $X(v)^o.f_i = \text{Min}\{X(v)^o.f_i, X(v).\phi^o_j \oplus f_{(x_j \rightsquigarrow X(v).anc_i)}\}$;
18 $X(v)^i.f_i = \text{Min}\{X(v)^i.f_i, f_{(X(v).anc_i \rightsquigarrow x_j)} \oplus X(v).\phi^i_j\}$;
19 $X(v)^o.f_{|X(v).anc|} = f_{min}$;
20 $X(v)^i.f_{|X(v).anc|} = f_{min}$;
21 Return $X(v)^o.f$, $X(v)^i.f$ and $X(v).pos$ for all $v \in V$;

Example 6 Consider the decomposition tree T_G as shown in Fig. 3 for the graph in Fig. 1. For vertex v_6, we have $X(v_6) = \{v_4, v_5, v_7, v_6\}$, $X(v_6).\phi^o = \{f_{(v_6,v_7)}, f_{(v_6,v_6)}\}$, $X(v_6).\phi^i = \{f_{(v_4,v_6)}, f_{(v_5,v_6)}, f_{(v_6,v_6)}\}$ and $X(v_6).anc = \{v_4, v_1, v_5, v_3, v_8, v_7, v_6\}$. Then, we can get $X(v_6).pos = \{1, 3, 6, 7\}$. Next, consider the computation of $X(v_6)^o.f_5$, i.e., the weight function for $(v_6 \rightsquigarrow v_8)$. Suppose the function arrays for all ancestors of $X(v_6)$ in T_G have been computed. Based on Lemma 1, we have $X(v_6)^o.f_5 = f_{(v_6,v_7)} \oplus f_{(v_7 \rightsquigarrow v_8)}$. To compute $f_{(v_7 \rightsquigarrow v_8)}$, since $X(v_6).pos_3 = 6 > 5$, we consider case 1. Therefore, we can derive that $f_{(v_7 \rightsquigarrow v_8)} = X((X(v_6).anc_6)^o.f_5)) = X(v_7)^o.f_5 = \{(0, 20), (60, 20)\}$, and $f_{(v_6,v_7)} = X(v_6).\phi^o_3 = \{(0, 6), (20, 12), (40, 6), (60, 6)\}$. Consequently, we can get $X(v_6)^o.f_5 = \{(0, 26), (20, 32),$

(40, 26), (60, 26)}. Similarly, for $X(v_6)^i.f_3$, i.e., the weight function for $(v_5 \rightsquigarrow v_6)$, we can get that $X(v_6)^i.f_3 = Min(f_{(v_5,v_6)}, f_{(v_5 \rightsquigarrow v_4)} \oplus f_{(v_4,v_6)})$. And we have $f_{(v_5,v_6)} = X(v_6).\phi_2^i$, and $f_{(v_4,v_6)} = X(v_6).\phi_1^i$. Then, to compute $f_{(v_5 \rightsquigarrow v_4)}$, since $X(v_6).pos_1 = 1 < 3$, we consider case 2. Hence, we get $f_{(v_5 \rightsquigarrow v_4)} = X(v_5)^o.f_1$, and $X(v_6)^i.f_3 = Min(f_{(v_5,v_6)}, f_{(v_5 \rightsquigarrow v_4)} \oplus f_{(v_4,v_6)}) = \{(0, 12), (20, 24), (60, 12)\}$.

Time Complexity Analysis. Consider the time complexity of Algorithm 2 for the tree decomposition. Firstly for the function preserved tree node formation (lines 3-8), each vertex takes $O(W^2 \cdot |S|)$ time to form new edges, and the time to order vertices costs $O(|V| \cdot log|V|)$, so it takes $O(|V|(W^2 \cdot |S| + log|V|))$. Regarding the tree formation (lines 9-12), it takes $O(|V| \cdot W)$ time to find the parent from each node's W vertices to form the tree. The sorting organization (lines 13-16) takes $O(|V|W \log W)$ time. Therefore, Algorithm 2 takes $O(|V|(W^2 \cdot |S| + log|V|)$ time. As for Algorithm 3, it takes $O(|V| \cdot W \cdot h \cdot |S|)$ time to compute the labels. Because $W \leq h - 1$, the total time complexity of TD-H2H construction is $O(|V|(W \cdot h \cdot |S| + log|V|))$.

Space Complexity Analysis. Our TD-H2H index structure mainly consists of the Position Array and Function Array. For each node $X(v)$ in T_G, $X(v).pos$ is a subset of $X(v).anc$, whose size is no larger than h, where h denotes the height of the decomposition tree. And $X(v).f$ stores all the time-dependent functions between v and the nodes in $X(v).anc$. Therefore, the space complexity of TD-H2H index is at $O(|V| \cdot h \cdot |S|)$.

4.2 Answering TD-*k*NN query

After the index construction, in this section, we introduce the algorithm to answer the TD-kNN query. Recall that in GLAD, the H2H index is served as the distance oracle for any pair of vertices in the graph. Similarly, in our solution, we compute the time-dependent fastest travel cost during TD-kNN query processing using the TD-H2H index. Consequently, let us first introduce the algorithm to compute the time-dependent fastest travel cost for any given pair of vertices.

4.2.1 Time-dependent fastest travel cost query

Given a pair of source and destination vertices s and d on $G(V, E, F)$, a departure time t, and the TD-H2H index, the major process to answer the time-dependent fastest travel cost query is to find out the LCA of $X(s)$ and $X(d)$ in the decomposition tree T_G, and then calculate the result based on its function arrays. The underlying rationale can be given by the following theorem.

Algorithm 4: Time-dependent Fastest Travel Cost (TFTC)

Input: $G(V, E, F)$, a source s, a destination d, time t and the H2H-Index
Output: $TFTC(s, d, t)$

```
1 X_L ← the LCA of X(s) and X(d) in T_G;
2 r ← +∞;
3 for all i ∈ X_L.pos do
4     tmp ← X(s)^o.f_{X_L.pos_i}(t) + X(d)^i.f_{X_L.pos_i}(t + X(s)^o.f_{X_L.pos_i}(t));
5     if tmp < r then
6         r ← tmp
7 Return r;
```

Theorem 3 *Given a pair of source and destination vertices s, d, the decomposition tree T_G of a road network G, and the corresponding TD-H2H index, let X_L be the LCA of $X(s)$ and $X(d)$ in T_G, we have:*

$$TFTC(s,d,t) = Min_{i \in X_L.pos}\{X(s)^o.f_{X_L.pos_i}(t) + X(d)^i.f_{X_L.pos_i}(t + X(s)^o.f_{X_L.pos_i}(t))\} \quad (3)$$

Proof According to the definition of TD-H2H index, suppose $X_L = \{w_1, w_2, \ldots, w_l\}$ sorted in ascending order of their positions in $X_L.anc$. Thus, we have $TFTC(s, X_L.anc_i, t) = TFTC(s, X(s).anc_i, t)$ for any $1 \leq i \leq |X_L.anc|$. Based on Theorem 1, we can get $TFTC(s, d, t) = \text{Min}_{v \in X_L}\{f_{(s \rightsquigarrow v)}(t) + f_{(v \rightsquigarrow d)}(t + f_{(s \rightsquigarrow v)}(t))\}$. As $X(s)$ and $X(d)$ are the decedents of X_L in T_G, we have $TFTC(s, d, t) = \text{Min}_{v \in X_L}\{f_{(s \rightsquigarrow v)}(t) + f_{(v \rightsquigarrow d)}(t + f_{(s \rightsquigarrow v)}(t))\} = \text{Min}_{i \in X_L.pos}\{f_{(s \rightsquigarrow X_L.anc_i)}(t) + f_{(X_L.anc_i \rightsquigarrow d)}(t + f_{(s \rightsquigarrow X_L.anc_i)}(t))\} = \text{Min}_{i \in X_L.pos}\{f_{(s \rightsquigarrow X(s).anc_i)}(t) + f_{(X(d).anc_i \rightsquigarrow d)}(t + f_{(s \rightsquigarrow X(s).anc_i)}(t))\} = \text{Min}_{i \in X_L.pos}\{X(s)^o.f_{X_L.pos_i}(t) + X(d)^i.f_{X_L.pos_i}(t + X(s)^o.f_{X_L.pos_i}(t))\}$. Thus, we finish the proof of this theorem. □

Based on Theorem 3, we can simply scan the function arrays $X(s)^o.f$ and $X(d)^i.f$ to find the weight functions $f_{(s \rightsquigarrow X(x).anc)}$ and $f_{(X(d).anc \rightsquigarrow d)}$, respectively, and calculate the travel cost with Equation 3. Algorithm 4 shows the pseudocode for computing the time-dependent fastest travel cost. In particular, we apply the same method in [28] to calculate the LCA in constant time. Then compute the cost based on Theorem 3. We give an example to show how the query process as follows.

Example 7 Consider the example in Fig. 1. Given a query $TFTC(v_7, v_0, 20)$, with a partial TD-H2H index as shown in Fig. 4, we first get $\text{LCA}(v_7, v_0) = X(v_8)$ in T_G. Since $X(v_8).pos = \{2, 3, 4, 5\}$, we have $TFTC(v_7, v_0, 20) = \text{Min}\{X(v_7)^o.f_2(20) + X(v_0)^i.f_2(20 + X(v_7)^o.f_2(20)), X(v_7)^o.f_3(20) + X(v_0)^i.f_3(20 + X(v_7)^o.f_3(20)), X(v_7)^o.f_4(20) + X(v_0)^i.f_4(20 + X(v_7)^o.f_4(20)), X(v_7)^o.f_5(20) +$

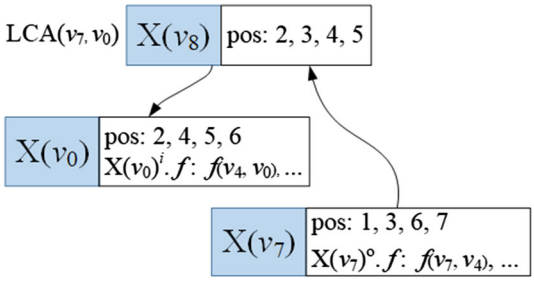


Fig. 4 Query processing for $TFTC(v_7, v_0, 20)$

$X(v_0)^i.f_5(20 + X(v_7)^o.f_5(20))\} = \text{Min}\{39 + 56, 25 + 72, 8 + 43, 20 + 12\} = 32$. Thus, the fastest travel time from v_7 to v_0 departing at 20 is 32.

Complexity Analysis. The time cost of TFTC query based on the TD-H2H index is mainly caused by the LCA query for two nodes in the decomposition tree T_G and the retrieval of the function arrays. According to [28], the computational cost for the LCA query is at $O(1)$. Since we have that the largest size of $X(v)$ in the decomposition tree is bounded by the width of the tree. Thus, the total computational cost of the TFTC query is bounded by $O(W)$, where W is the tree width.

4.2.2 TD-kNN query

Next, we introduce the algorithm to answer TD-kNN query. As aforementioned, our framework is similar to that in GLAD while replacing the H2H index with the TD-H2H index. Nevertheless, to further improve the query efficiency, in our work, we propose two pruning strategies as follows.

Lower-bound Cost Pruning. Based on Definition 5, we observe that if the lower bound travel cost from an object o_a to the query point q is larger than the exact time-dependent fastest travel cost from another object o_b to q, then o_a can be pruned by o_b regardless of departure time.

Observation 1 *Given two objects o_a, o_b, a departure time t, and a query point q, if $LBC(o_a, q) > TFTC(o_b, q, t)$, then we have $TFTC(o_a, q, t') > TFTC(o_b, q, t)$ $\forall t' \in [t_s, t_e]$.*

Motivated by this observation, in our work, we build an H2H index on the lower bound graph (Ref. Definition 4). Then, after retrieving the first k candidate objects with the corresponding $TFTC$ values, during the further exploration to discover other objects, we can first calculate the lower bound travel cost for them based on the H2H index, to see whether they can be pruned by the k-th candidate object we have found. In this way, unnecessary computation of time-dependent travel cost for some objects can be avoided, which can significantly show in Sect. 6.

Label Pruning. Recall that according to Theorem 3, the computation of time-dependent fastest travel cost for two vertices s, d actually contains two portions: the travel cost from s to the label in its LCA nodes in the decomposition tree, and the cost from this label to d. We observe that if two objects pass through the same label, we can omit the second part of computation for one of the labels based on the following observation.

Observation 2 *Given two objects o_a and o_b, a query point q with a departure time t, let v be a common label of them, i.e., $v \in LCA(X(o_a), X(q)) \cap LCA(X(o_b), X(q))$. If $TTC(o_a, v, t) > TTC(o_b, v, t)$, based on the FIFO property, we have $TTC(o_a, v, t) + TTC(v, q, (t + TTC(o_a, v, t))) > TTC(o_b, v, t) + TTC(v, q, (t + TTC(o_b, v, t)))$. Note that TTC is calculated by the aggregated weight functions as shown in Theorem 3.*

Consequently, consider an object o_f, which has been filtered (this object will never appear in the result set). Suppose that we have another object o_c to be examined and we get $v \in LCA(X(o_f), X(q)) \cap LCA(X(o_c), X(q))$, with $TTC(o_c, v, t) > TTC(o_f, v, t)$, then based on Observation 2 we can omit the computation of $TTC(v, q, (t + TTC(o_c, v, t))$ since the total cost must be larger than that of o_f. Motivated by this, during query processing, we maintain a buffer for each vertex v to record the smallest travel cost from all the current filtered objects to v at the query time, i.e., $\min_{o_i \in O_f} TTC(o_i, v, t)$, where O_f is the set of objects being filtered at the current stage of query processing. During query processing, we check this buffer to see whether we can avoid the second part of computation for some objects. And we update this data structure after the examination of each candidate object.

Example 8 Consider the example in Fig. 5, we assume that v_1 and v_2 are the common labels in LCA(o_1, q), LCA(o_2, q) and LCA(o_3, q). Suppose that o_1 and o_2 have been filtered and we have $TTC(o_1, v_1, t) = 20$ and $TTC(o_2, v_1, t) = 10$. Then, we set the buffer value for $v_1.cost$ as 10. Given $TTC(o_3, v_1, t) = 20$, which is larger than $v_1.cost$, we can omit the computation of $TTC(v_1, q, t + TTC(o_3, v_1, t))$. Similarly, given $TTC(o_3, v_2, t) = 15$, we can omit $TTC(v_2, q, t + TTC(o_3, v_2, t))$.

Query Algorithm. The pseudo-code of the TD-kNN query algorithm is shown in Algorithm 5. We start from the grid of

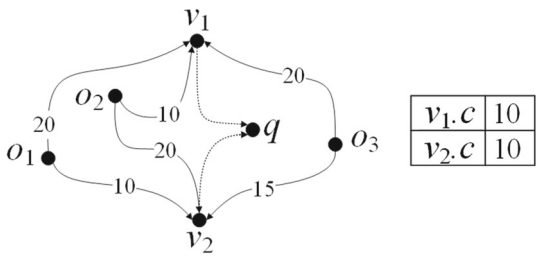


Fig. 5 Label pruning

Algorithm 5: TD-kNN

Input: Road network $G(V, E, F)$, query point q, departure time t, grid index L, lower bound cost scheme LBC, TD-H2H index
Output: TD-kNN of q

1 Let h be the grid containing q, $H \leftarrow \phi$, $NH \leftarrow h$;
2 Let C be the candidate set of the kNN queries;
3 Let UB be the upper bound of the travel time of the k-th nearest neighbor to q and initially set to $+\infty$;
4 Let $LB \leftarrow 0$ and $T(o, q)$ denotes the current travel cost from object o to q, $T(o, q) \leftarrow +\infty$;
5 $\forall v \in V$, $v.c \leftarrow +\infty$;
6 **while** $UB > LB$ **do**
7 let $L(NH)$ be the set of objects in NH;
8 **foreach** $o \in L(NH)$ **do**
9 **if** $C.size() < k$ **then**
10 $C \leftarrow o$;
11 **else**
12 Let T_k be the k-th largest travel cost of objects in C;
13 **if** $LBC(o, q) \leq T_k$ **then**
14 **foreach** $v \in LCA(o, q)$ **do**
15 **if** $TTC(o, v, t) \leq v.c$ **then**
16 $tmp \leftarrow TTC(o, v, t) + TTC(v, q, t + TTC(o, v, t))$;
17 $T(o, q) \leftarrow \min\{T(o, q), tmp\}$;
18 **if** $T(o, q) < T_k$ **then**
19 $C \leftarrow o$, update T_k;
20 **else**
21 **foreach** $v \in LCA(o, q)$ **do**
22 $v.c \leftarrow \min\{v.c, TTC(o, v, t)\}$;
23 $H \leftarrow H \cup NH, UB \leftarrow T_k$;
24 Let NH be the set of grids that are neighbors of H but excluding the grids in H;
25 Let D_L be the minimum Euclidean distance from q to the edge of NH, $LB \leftarrow D_L / speed_{max}$;
26 **return** top$-k$ objects in C with fastest travel time;

size 1×1, and then gradually expanding the grid to 3×3, 5×5,…until the kNN candidates are found. Then, during the further exploration, we set an lower-bound cost to be the cost travel from the boundary of explored grid to the query point with the maximum speed admitted for the whole road network. Lines 1–5 illustrate the initialization of this algorithm. We maintain a set C to record the candidate objects, and a set H for the grids to explore. Let NH to be neighbor grids surrounding H, and initially set to be $\{h\}$, which is the grid where q is located. We use UB to indicate upper-bound cost of the k-th nearest neighbor to q and LB as the lower-bound as aforementioned. For each object o, we maintain a $T(o, q)$ to denote the current travel cost from o to q, and for each vertex v, we use $v.c$ to indicate the buffer for the smallest cost from all the current filtered objects to v. Then, we start the iteration to explore the grid and retrieve candidate objects. At the beginning of each iteration, we check whether the current upper-bound value has exceed the lower-bound, and later we update both to narrow down the gap of them to accelerate the convergence. In Lines 7–10, we calculate the travel cost of all moving objects in NH to the query location q using TD-H2H index, and add these objects to the candidate set C until we get k objects. Let T_k be the k-th largest cost among those in C. In Line 13, we calculate the lower bound cost $LBC(o, q)$ with the H2H index, and check whether $LBC(o, q)$ is larger than T_k. If so, we can prune this object based on Observation 1. Otherwise, in Lines 14–17, for each $v \in LCA(o, q)$, we first calculate the travel cost $TTC(o, v, t)$ and check $v.c$ to see whether we can omit the further computation based on Observation 2. Alternatively, in Lines 18–22, we calculate the exact travel cost for this object and update C and T_k accordingly. After this iteration, in Lines 23–25, we update H to include the grids in NH and update NH by exploring the next surrounding grids. And we update the upper-bound and lower-bound cost based on the current results. At the end, we return the top-k objects with the fastest travel cost in C as the TD-kNN query answer.

Complexity Analysis. The time complexity for the TD-kNN query is based on two parts: the cost of grid expansion and the cost TFTC query for each object. Though we propose several pruning strategies, at the worst case, for each object we need to compute both the TTC and TFTC query, each of which is bounded by $O(W)$ as analyzed in Sect. 4.2.1. Thus, the overall computational cost for the TFTC part is bounded by $O(W|M_{avg}|)$, where $|M_{avg}|$ denotes the average number of objects retrieved during query processing. Note that in the worst case, $|M_{avg}|$ could be up to $|M|$, which is the total number of objects. As for the grid expansion, the cost is bounded by the number of objects to retrieve, which is dominated by the TFTC part. Consequently, the time complexity for our TD-kNN query is at $O(W|M_{avg}|)$. In contrast, the time complexity of TD-kNN query for TD-Dijkstra and TD-G-tree is at $O(|M_{avg}|(|V| \cdot log|V| + |E|))$ and $O(|M_{avg}|(log_2{}^2 k_f \cdot |V| \cdot log_2{}^2 |S|))$, respectively.

5 Extension on TD-kNN query

Applications of TD-kNN query can be found in the taxi-hailing service, where each moving object represents a taxi running on the road network. When a user proposes a taxi request, the system finds the k nearest taxis and assigns one of them to serve the user based on some specific strategies. In this case, we consider all the moving objects available to serve. However, in practice, some of the taxis might have been occupied by existing users, and these occupied objects might arrive at their destinations soon so that they could also approach the query point in a short time. In [42], the authors propose the Approachable k Nearest Neighbor

(AkNN) query, which takes into consideration of both occupied and unoccupied moving objects, e.g., taxi. We give such objects a specific definition as follows.

Definition 12 *(Conditional Moving Object)* We define the conditional moving object o^c on the time-dependent road network as a tuple, i.e., $o^c = (p_c, p_d, c)$, where p_c (resp. p_d) indicates the current (resp. the imminent destination) location of this moving object, and c represents the condition (occupied/non-occupied). We have $c = 1$ to represent "occupied" and $c = 0$ for "non-occupied" condition. Note that $p_c = p_d$ if $c = 0$.

Given a set of non-occupied/occupied moving objects M^c and a query point on the time-dependent road network, we aim to return the k nearest objects those can earliest response to the request after they finish their current journey (arrive at the destinations). These objects are regarded as the imminent approachable candidates to serve the requests. Thus, we calculate the travel cost from a conditional moving object to the query point by summarizing the cost from its current location to its coming destination and the cost from its coming destination to the query point. Formally, we define the time-dependent approachable kNN as follows.

Definition 13 *(Time-dependent Approachable kNN (TD-AkNN))* Given a query point q, a set M^c of moving objects on a time-dependent road network $G(V, E, F)$, a departure time t, and an integer $k \leq |M^c|$, time-dependent approachable kNN returns a subset $R \subseteq M^c$ of k moving objects such that for all $o_i^c \in M^c \setminus R$, $TFTC(o_i^c.p_c, o_i^c.p_d, t) + TFTC(o_i^c.p_d, q, (t + TFTC(o_i^c.p_c, o_i^c.p_d, t))) \geq TFTC(o_j^c.p_c, o_j^c.p_d, t) + TFTC(o_j^c.p_d, q, (t + TFTC(o_j^c.p_c, o_j^c.p_d, t)))$ for any $o_j^c \in R$.

To address the TD-AkNN query, we adopt the same framework like the one for the TD-kNN query. Note that in this case, each moving object contains two locations. In our work, we maintain the grid index only based on the destination location. Moreover, we regard the time-dependent fast travel cost computation based on the TD-H2H index as a black-box. Particularly, the pseudo-code of the algorithm for TD-AkNN query is shown in Algorithm 6. Lines 1–4 illustrates the same initialization as the ones for the TD-kNN Algorithm 5. We also maintain a set C to hold candidate objects. In Lines 6–9, for the object o^c with $o^c.p_d \in NH$, we calculate the travel cost of path $\{o^c.p_c, o^c.p_d, q\}$ with TD-H2H index, and add them to C until we get k candidate objects. Then, we set the k-th largest travel cost among the objects in the candidate set C as T_k in Line 11. In Line 12, before we calculate the fastest travel cost from o^c to q, we first calculate the lower bound cost of path $\{o^c.p_c, o^c.p_d, q\}$. If the lower bound cost is larger than T_k, we do not need to calculate the travel cost for o^c. Otherwise, in Line 13, we calculate the travel cost tmp of path $\{o^c.p_c, o^c.p_d, q\}$. If tmp is smaller than T_k, which means o^c can be a candidate object, then we update the candidate set C and T_k accordingly in Lines 14-15. After that, we use the same method as that in TD-kNN to update H, UB, NH, and LB. Finally, we return the top-k objects with the fastest travel cost in C as the TD-AkNN query answer.

Algorithm 6: Grid-base TD-AkNN

Input: Road network $G(V, E, F)$, lower bound cost scheme LBC, query point q, grid index L, query time t
Output: TD-AkNN of q

```
1  Let h be the grid containing q, H ← φ, NH ← h;
2  Let C be the candidate set of the kNN queries;
3  Let UB be the upper bound of the travel time of the k-th nearest
   neighbor to q and initially set to +∞;
4  Let LB ← 0;
5  while UB > LB do
6      Let L(NH) be the set of objects in NH;
7      foreach o^c.p_d ∈ L(NH) do
8          if C.size() < k then
9              C ← o^c;
10         else
11             Let T_k be the k-th largest travel cost of objects in C;
12             if LBC(o^c.p_c, o^c.p_d) + LBC(o^c.p_d, q) ≤ T_k then
13                 tmp ← TFTC(o^c.p_c, o^c.p_d, t) +
                   TFTC(o^c.p_d, q, t + TFTC(o^c.p_c, o^c.p_d, t));
14                 if tmp < T_k then
15                     C ← o^c, Update T_k;
16     H ← H ∪ NH, UB ← T_k;
17     Let NH be the set of grids that are neighbors of H but
       excluding the grids in H;
18     Let D_L be the minimum Euclidean distance from q to the
       edge of NH, LB ← D_L/speed_max;
19 Return top−k objects in C with fastest travel time;
```

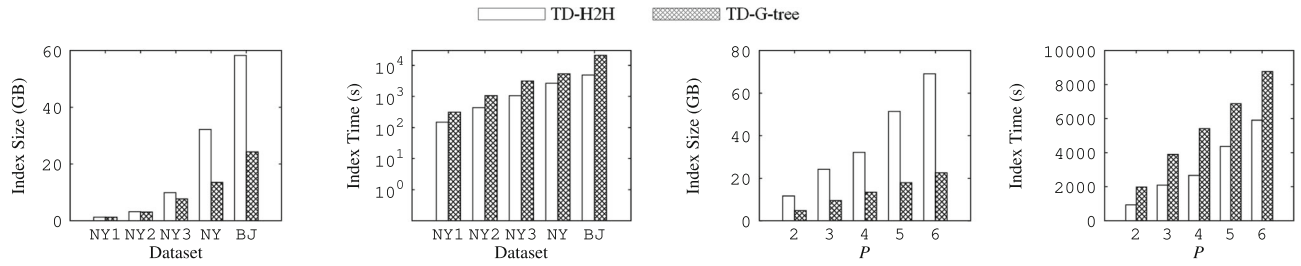
Complexity Analysis. Since Algorithm 6 searches the objects in a similar way to Algorithm 5, they have the same time complexity, which is $O(W|M_{avg}|)$. However, the calculation of the lower bound cost and TFTC of each moving object needs to be done twice for the TD-AkNN query due to the occupancy, so the query time will be longer than performing a TD-kNN query.

6 Experimental study

In this section, we present the experimental study to show the performance of our proposed solutions for the time-dependent k nearest neighbor queries. We first introduce the experimental setting, followed by the results with various parameter settings.

6.1 Experimental settings

Datasets: We conduct our experiments on real road network from New York city (NY) and Beijing city (BJ). To



(a) Index Size Varying Datasets **(b)** Index Time Varying Datasets **(c)** Index Size Varying p **(d)** Index Time Varying p

Fig. 6 Performance of index construction

Table 2 Parameter settings

Parameter	Description	Values
k	Input parameter of kNN	10, 20, 30, 40, 50
$\|M\|$	Number of moving objects	1, 2, 3, 4, 5($\times$ 10000)
L	Length of each grid(m)	500, 1000, 1500, 2000, 2500
p	Number of interpolation points	2, 3, 4, 5, 6
θ	Proportion of occupied moving objects	0%, 30%, 60%, 90%

evaluate the scalability of our algorithms, we extract three sub-networks from NY, following the same idea in TOAIN [5]. Specifically, we gradually expand the road network from the center of New York city to get a certain number of vertices and the corresponding edges. We denote these three sub-networks as NY1, NY2, and NY3. Table 3 shows the sizes of vertices and edges for different datasets, and NY is the default dataset. The height and width of the decomposition tree of these datasets are also shown in Table 3. It is worth noting that although NY and BJ have similar numbers of vertices, the distribution of vertices in BJ is much denser. As a result, the height and width of the decomposition tree are larger than those of NY, which is in line with the results in [31].

Queries: We randomly choose 10,000 vertices as the query points, and randomly select a departure time for each of them within the whole time range of [0,1440]. In particular, for the number of interpolation points, the default value is 4, which are (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4). Next we will introduce how to generate the piecewise linear weight function for each edge. Ideally, the weight functions could be extracted from a set of trajectories. However, since we do not have the corresponding trajectory data, we manually simulate the weight functions based on common sense of the road condition. Firstly, we set the x coordinate of the interpolation points as follows: $x_1 = 0$, $x_4 = 1440$, $x_2 \in [510, 570)$, and $x_3 \in [990, 1070)$, where x_2 and x_3 are randomly selected from the corresponding range. As for the y coordinate, we set three levels of speed values with $sp_1 = 1000$m/minute, $sp_2 \in [500, 900]$m/minute, and $sp_3 \in [300, 750]$m/minute. Then, for each edge (u, v), let $l(u, v)$ be the length of this edge. We set $y_i = l(u, v)/sp_i$, for $1 \leq i \leq 3$, and $y_4 = y_3$. The intuition is that during mid-night, the traffic flow is smooth and the vehicle usually can have free flow speed, while during day time there could be some deduction due to the increase of traffic flow. Table 2 shows all the parameter settings in our experimental study, where the default values are marked in bold.

Table 3 Real-world maps and tree parameters

Dataset	# Vertices	# Edges	h	W
NY1	32527	90710	237	91
NY2	66581	184464	340	120
NY3	134064	396182	457	136
NY	264346	733846	504	141
BJ	265348	688962	716	320

Methods: As discussed in Sect. 4, there is no existing solutions for TD-kNN query. But our proposed framework for both TD-kNN and TD-AkNN can be easily integrated with any existing time-dependent fastest travel cost (TFTC) query algorithm. In order to evaluate the efficiency of the designed framework as well as the superiority of our proposed TD-H2H index, we apply the same framework to incorporate the TD-G-tree index and TD-Dijkstra algorithm as our competitive solutions. We simply denote these algorithms as TD-H2H, TD-G-tree, and TD-Dijkstra, respectively. In other words, the above three algorithms use the same framework but invoke different TFTC query algorithms to calculate the time-dependent travel cost from an object to the query point.

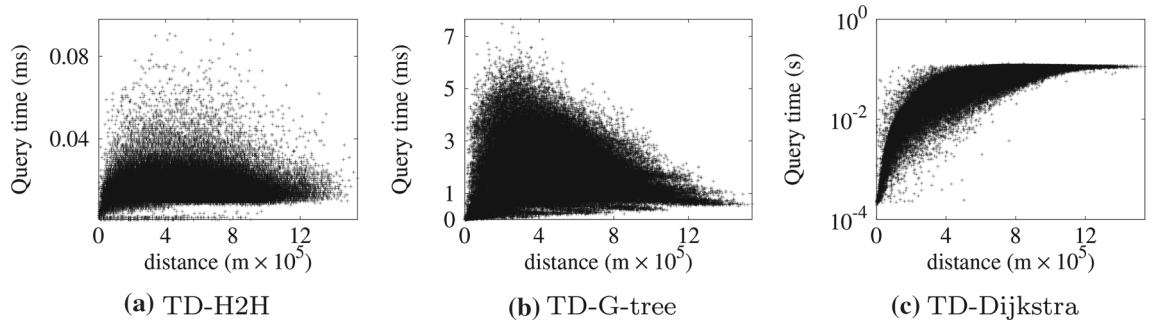


Fig. 7 TFTC query with different road network distance

Implementation details: All algorithms[1] are implemented in C++ and compiled with GNN GCC with full optimization. The codes executes on an Intel(R) Xeon(R) W-2245 CPU @3.2GHz with 250GB RAM under Linux (Ubuntu 18.04 LTS, 64bit).

6.2 Experimental results

6.2.1 Index construction

We first compare the performance of the proposed TD-H2H index with the existing TD-G-tree under different datasets and different numbers of interpolation points for the weight functions. Recall that the construction of the TD-H2H index involves tree decomposition of the graph, calculating the time-dependent weight functions between each vertex in the decomposition tree and its ancestors, and storing them in an array of weight functions. By contrast, the TD-G-tree index involves partitioning the graph into a balanced tree structure, calculating the weight functions between boundary vertices and between the non-boundary vertices and boundary vertices, and storing these functions in matrices on the non-leaf nodes and leaf nodes in the tree. Figure 6 shows the experimental result of index construction in comparison with the TD-G-tree. As we can see, TD-H2H needs more space for weigh functions than TD-G-tree, while taking less time to construct the index at all settings. The index size of TD-H2H is about 3 times that of TD-G-tree, and this is because TD-H2H maintains the weight function array for every vertex, while TD-G-tree only stores them for a small set of vertices, e.g., the boundary vertices. However, with improved storage and memory hardware, an index size at about 30GB is acceptable for a whole NY road network. Since TD-H2H computes the weight functions in a top-down manner, it reuses the existing information to reduce the overall computational cost.

Effect of network size. It is obvious that the index size and construction time will increase as the road network becomes larger, as shown in Fig. 6a and b. The size of TD-G-tree grows more slowly than TD-H2H, because its boundary vertices do not increase dramatically with the growth of network size, while the index size of our TD-H2H increases faster since the more vertices, the heavier weight functions required. The index construction efficiency of TD-H2H is less affected by the size of the network compared with the TD-G-tree, which also benefits from the re-usage of partial information from the ancestors of a node in the decomposition tree. In addition, the index for BJ road network requires larger space and more construction time compared to NY, though they have the similar number of vertices. This is mainly because the topological structure in BJ road network is more complex than that of NY, resulting in larger height and width in the decomposition tree as illustrated in Table 3.

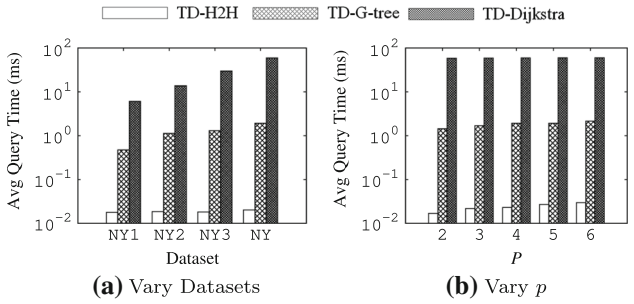
Effect of the number of interpolation points. As illustrated in Fig. 6c and d, the index size and construction time of both indices increase as the size of interpolation points increases. That is because the number of the interpolation points of the aggregated weight function of a path will increase exponentially when aggregating more weight functions, which makes sense, and the impact for both indices is similar to the scenario of varying map size.

6.2.2 Fastest travel cost query processing

As introduced in Sects. 4.2 and 5, the TFTC query will be invoked multiple times in order to answer a TD-*k*NN or TD-A*k*NN query. Thus conduct a set of performance comparisons among the proposed TD-H2H index and the existing TD-G-tree index and TD-Dijkstra algorithm in supporting the TFTC query.

Effect of the source-destination distance. We randomly generate 100K pairs of source and destination points under *NY* dataset, and set the departure time randomly. The TFTC query is issued for each pair, and the corresponding response time is recorded. Then, the shortest road network distances of these 100K pairs on the road network are calculated, respectively, and divided into 4 intervals by distances, i.e.,

[1] The source code, data, and/or other artifacts have been made available at https://github.com/jiajia4487/TD-H2H-kNN.



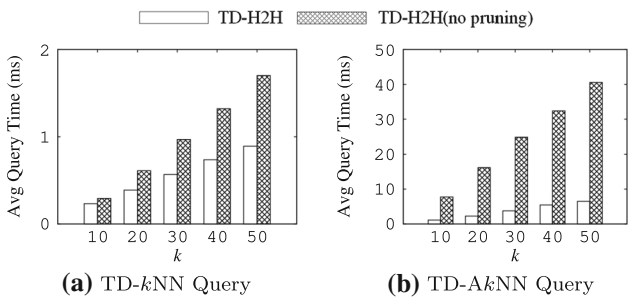
(a) Vary Datasets **(b)** Vary p

Fig. 8 TFTC query processing time

Table 4 Pruning power varying k

k	10	20	30	40	50
P1 (NY)	72.5%	70.9%	70.2%	69.8%	69.4%
P2 (NY)	66.5%	70.3%	72.6%	73.7%	74.4%
P1 (BJ)	24.2%	16.9%	15.2%	15.9%	12.4%
P2 (BJ)	25.2%	22.9%	27.9%	30.2%	32.5%

$(0, 4, 8, 12) \times 10^5$m. Figure 7 shows the query time distribution for different solutions, where each point indicates the query time of one request. As we can see, with the increase in road network distance, the query time of TD-Dijkstra increases dramatically. This is because TD-Dijkstra expands the graph from the source point until it reaches the destination, whose query efficiency is naturally related to distance. TD-G-tree shows more stable performance than TD-Dijkstra, because it skips a large number of unnecessary vertices. Nevertheless, when the two points are far away, and their least common ancestor (LCA) is close to the root node, the number of boundary points that need to be verified increases, which will also lead to a deduction of efficiency. As for the TD-H2H index, its efficiency depends more on the size of the array in the LCA, which is related to the contraction order of this LCA rather than the distance. Consequently, it is least affected by the distance, which means our solution is more robust to the TD-kNN query with various underlying road network distances.



(a) TD-kNN Query **(b)** TD-AkNN Query

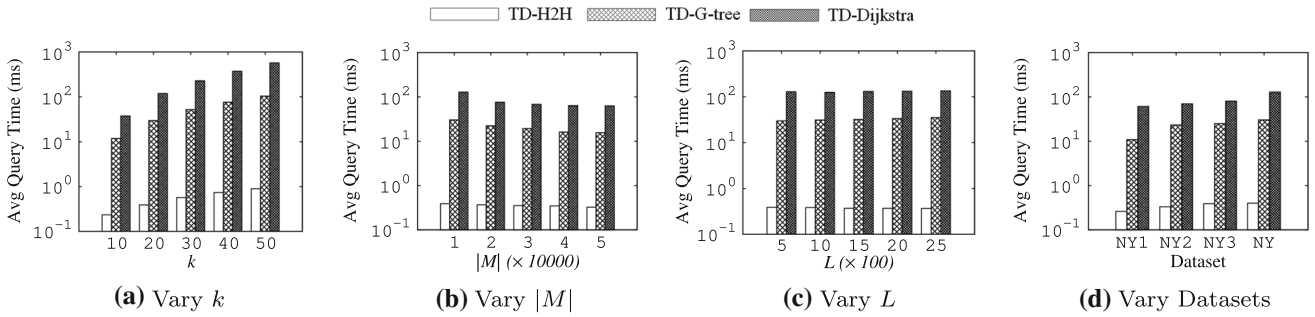
Fig. 9 Query processing with two pruning strategies

Effect of network size and number of interpolation points. Figure 8 illustrates the average running time for answering TFTC query of three algorithms under different datasets and different numbers of interpolation points for each edge. It can be seen that when the network size increases, the query response time of the three algorithms all increase. TD-Dijkstra increases the fastest because in a larger road network, the source and destination point might be farther , and more vertices need to be traversed. As for TD-G-tree and TD-H2H, a larger network indicates a deeper decomposition tree, and the growth of boundary points (in TD-G-tree) and associated vertices (in TD-H2H) in LCA will increase the query time. In terms of the interpolation points, the aggregated weight functions pre-computed for the matrices in TD-G-tree and the arrays in TD-H2H will increase greatly, resulting in a longer lookup time when calculating the travel cost. In contrast, TD-Dijkstra does not concatenate the weight functions but directly calculates the arrival time while exploring each edge. Hence, the increased lookup time is less, so it is less affected by the number of interpolation points. As we can see, the TD-H2H always beats the other two solutions in orders of magnitude.

6.2.3 Effectiveness of pruning strategies

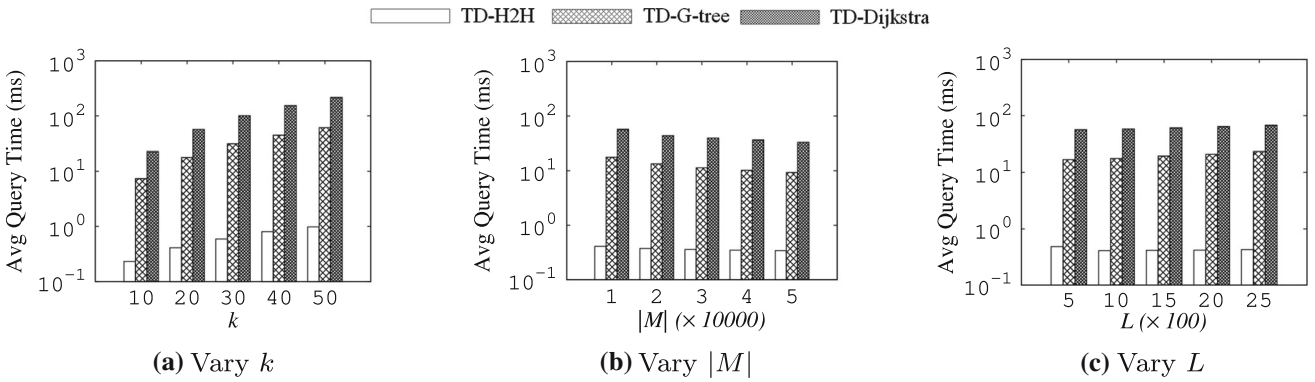
We evaluate the effectiveness of the pruning strategies based on Observation 1 and 2, and we denote these two strategies as P1 and P2, respectively. We first conduct a case study to see how well these two pruning strategies work. In particular, we perform 100K queries and keep track of both the numbers of objects that have been visited and the numbers of TTC computations of each query processing for evaluating the pruning power of P1 and P2, respectively. That is because P1 helps to filter those unpromising objects, while P2 helps to omit the second part of computation for some vertices in an LCA of the examined object and query point. For each setting, we conduct the same set of queries twice and record the sizes of traversed objects and calculated TTC, respectively, so that we can obtain the pruning power. Formally, we define the number of objects visited with (resp. without) pruning as O_p (O_{np}). Then, the pruning power is calculated by $(O_{np} - O_p)/O_p$ for P1. And the number of TTC computations is used instead of the number of objects visited in the calculations for P2. Table 4 shows the results of pruning power while varying k under the maps of New York and Beijing, where P1 means we only apply the first strategy.

As we can see from the interesting results, as the value of k grows, the trends of the pruning power of P1 and P2 are opposite under both road networks. However, the performance of P1 and P2 is very different regarding different road networks. As described in Sect. 4.2.2, if the lower bound cost of the object is larger than T_k, it can be pruned safely, where T_k represents the cost of the k-th moving object found so far.



(a) Vary k **(b)** Vary $|M|$ **(c)** Vary L **(d)** Vary Datasets

Fig. 10 TD-kNN query processing on NY dataset



(a) Vary k **(b)** Vary $|M|$ **(c)** Vary L

Fig. 11 TD-kNN query processing on BJ dataset

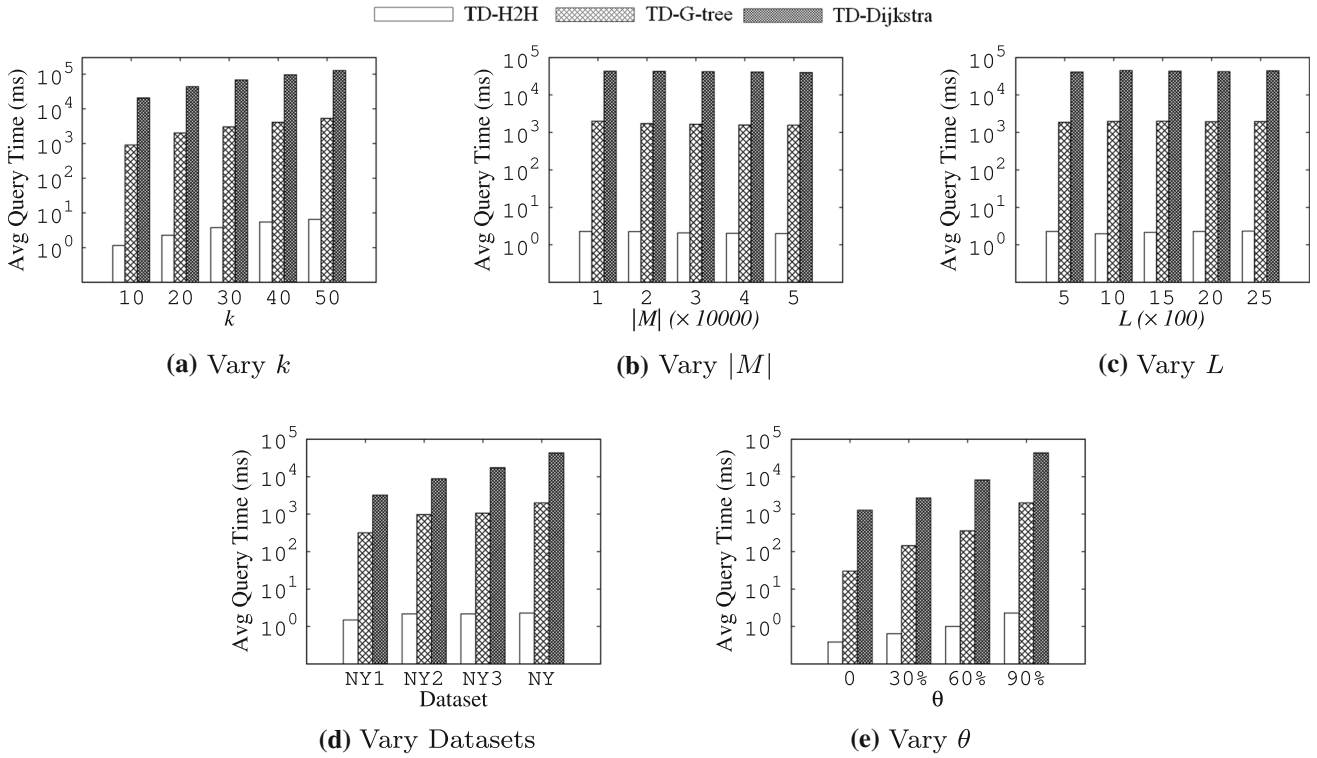
Obviously, a smaller value of T_k means that more objects can be filtered out. As k increases, it needs to search larger space to obtain final results and then T_k increases accordingly. Hence, the pruning power of P1 becomes weaker. Regarding P2, with more objects to verify, we might find more objects sharing the same LCA. Therefore, as k increases, the pruning power of P2 turns out to be stronger. As for the performance of the pruning power under different road networks, it is closely related to the properties of road networks. As mentioned in Table 3, although both road networks have a similar number of vertices, the lengths of road segments in BJ are generally shorter than those in NY. Since the length of the road segment is taken into account when generating the time-dependent functions, the change in the travel time of the shorter road segment is smaller. Therefore, when utilizing the lower bound of travel time for pruning by P1, fewer edges can be filtered out in BJ, so the performance of P1 is worse than that in NY as shown in Table 4. P2 also performs worse in BJ, since the decomposition tree of BJ has a larger width, so that there are fewer vertices passing through the same LCA.

Next, we evaluate the combination of the pruning power by reporting the query processing time with or without the pruning strategies. Figure 9 shows the results varying the value of k. As we can see, the query time without pruning nearest doubles that with pruning for TD-kNN, and the pruning advantage is more significant in answering TD-AkNN.

6.2.4 Efficiency of TD-kNN query

We compare the TD-kNN query efficiency of three solutions by varying four different parameter, i.e., k, object size $|M|$, gird length L, and network size. Figures 10 and 11 show the results of four sets of experiment on the maps of NY and BJ, respectively, each of which is the average query time over 100K queries. For all settings, our solution achieves 2-3 orders of magnitude faster than the TD-G-tree and TD-Dijkstra.

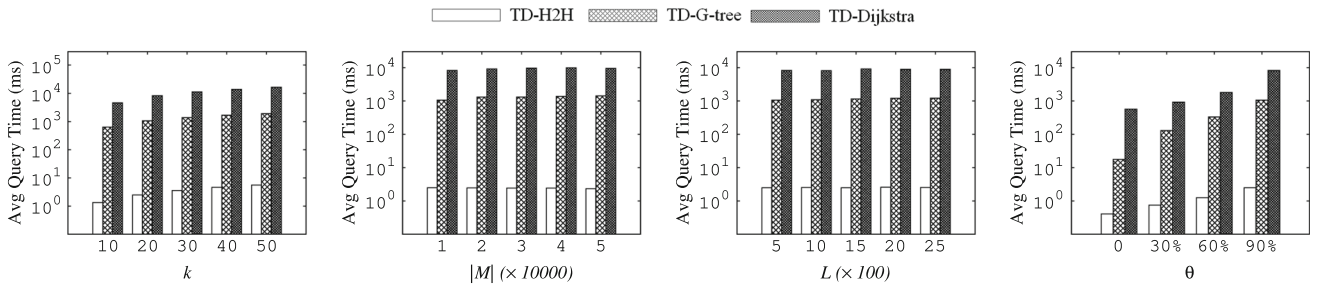
From Fig. 10a, we can see that the query time increases as k increases for all solutions, as the larger k indicates more exploring space to retrieve moving objects, thus higher computational cost. As for the object size, the query time of all algorithms decrease slightly, as shown in Fig. 10b. This trend is similar to the impact of the road network distance on the TFTC query. The underlying intuition is that the increase of $|M|$ leads to a higher density of moving objects. Thus, k objects can be found quicker, and the exploration space from the query point to its kNN is smaller. As illustrated in Fig. 10c, L has little impact on the query time of all algorithms for TD-kNN query. Intuitively, the smaller L, the more grid levels need to be explored, and vice versa. However, bene-



(a) Vary k **(b)** Vary $|M|$ **(c)** Vary L

(d) Vary Datasets **(e)** Vary θ

Fig. 12 TD-AkNN query processing on NY dataset



(a) Vary k **(b)** Vary $|M|$ **(c)** Vary L **(d)** Vary θ

Fig. 13 TD-AkNN query processing on BJ dataset

fiting from the pruning strategies, the influence of L reduce for retrieving the candidate objects. In terms of network size, when the network becomes larger, the query time increases very slightly for three solutions as illustrated in Fig. 10d. The size of network is closely related to the height and width of both tree structures generated by TD-G-tree and TD-H2H, thus increasing the overall query time. The TD-Dijkstra algorithm is very sensitive to the exploration space. With larger network and randomly generated query, there could be some queries requiring larger exploration space, thus resulting in higher computational cost.

As it can be seen from Fig. 11, the effect of the three parameters on all three algorithms for TD-kNN query is similar under the road network of BJ to that of NY. Specifically, the query time of all solutions on BJ also increases as k increases. Meanwhile, L has little impact on the query time of all algorithms. As for M, our algorithm tends to be stable when M increase, while the other two solutions go down slightly. Overall, TD-H2H is 2–3 orders of magnitude faster than TD-G-tree and TD-Dijkstra in nearly all settings, which verifies the scalability and robustness of our proposed algorithm.

6.2.5 Efficiency of TD-AkNN query

Next, we evaluate the efficiency of the TD-AkNN query. Similar to the ones for the TD-kNN query, we consider the impact of different parameters. Except for the aforementioned four parameters, in the TD-AkNN query, we have one more set

of experiment that varies the proportion of occupied objects, i.e., θ.

Figure 12 shows the performance results for the TD-AkNN query with three solutions under the dataset of NY. Remarkably, our solution is always superior to the competitors in 3-4 orders of magnitude in all settings. The impacts of k, $|L|$, and network size are similar to the ones for the TD-kNN query, which shows the robustness of our design framework and the proposed TD-H2H. Differently, the query time of TD-AkNN of three algorithms is very little affected by $|M|$ on both of the maps, as shown in Fig. 12b. The reason is that although the destination of the occupied object is closer to the query point, its current location might be far away. Thus, the impact of object density has little correlation with the road network distance from object to query point. As for θ, Fig. 12e illustrates the query time of the three algorithms under different occupancy rates. For an occupied moving object, to calculate the travel cost to the query point, both the travel time from its current location to its destination and this destination to the query point need to be calculated. Consequently, when we increase the number of occupied moving objects, the overall computational cost increases accordingly.

We also conduct the experimental study of TD-AkNN on the road network of BJ for the three algorithms, and the results are illustrated in Fig. 13. It can be seen that the query time of all solutions increases with the value of k and θ, while it is not greatly affected by $|M|$ and $|L|$. Similarly, our proposed TD-H2H still performs well in BJ road network, achieving 3-4 orders of magnitude faster against the TD-G-tree and TD-Dijkstra algorithms.

7 Conclusion

In this paper, we present a comprehensive study on TD-kNN queries, which aims to return k moving objects that can fastest arrive at a given query location departing at time t. We develop an efficient framework derived from the GLAD that includes a novel TD-H2H index for fastest travel cost computation and two pruning strategies to speed up TD-kNN query. We further extend our solution to address the TD-AkNN, where both occupied and non-occupied moving objects are considered. Extensive experiments on the real-world road network show that our solution is several orders of magnitude faster than the competitors.

Acknowledgements The research work described in this paper was partially supported by the Natural Science Foundation of Liaoning Education Department under Grant No. LJKZ0205 and was partially conducted in the JC STEM Lab of Data Science Foundations funded by The Hong Kong Jockey Club Charities Trust.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions

References

1. Didichuxing. https://gaia.didichuxing.com/
2. Uber. https://www.uber.com/
3. Yuan, J., Zheng, Y., Xie, X., Sun, G.: Driving with knowledge from the physical world. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 316–324 (2011)
4. Yuan, N.J., Yu, Z., Zhang, C., Xie, W., Xing, X., Sun, G., Yan, H.: T-drive: driving directions based on taxi trajectories. In: ACM SIGSPATIAL GIS 2010;ACM SIGSPATIAL International Conference on Advances in Geographic Information systems (2011)
5. 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. **11**(5), 594–606 (2018)
6. He, D., Wang, S., Zhou, X., Cheng, R.: Glad: a grid and labeling framework with scheduling for conflict-aware knn queries. IEEE Trans. Knowl. Data Eng. **99**, 1–1 (2019)
7. He, D., Wang, S., Zhou, X., Cheng, R.: An efficient framework for correctness-aware knn queries on road networks. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE) (2019)
8. Shen, B., Zhao, Y., Li, G., Zheng, W., Qin, Y., Yuan, B., Rao, Y.: V-tree: Efficient knn search on moving objects with road-network constraints. In: Data Engineering (ICDE), 2017 IEEE 33rd International Conference on, pp. 609–620. IEEE (2017)
9. 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. **27**(8), 2175–2189 (2015)
10. Ding, B., Yu, J.X., Qin, L.: Finding time-dependent shortest paths over large graphs. In: Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology, pp. 205–216 (2008)
11. Kanoulas, E., Yang, D., Tian, X., Zhang, D.: Finding fastest paths on a road network with speed patterns. In: 2006 IEEE 22nd International Conference on Data Engineering (ICDE), p. 10 (2006)
12. Gendreau, M., Ghiani, G., Guerriero, E.: Time-dependent routing problems: a review. Comput. Oper. Res. **64**, 189–197 (2015)
13. Wang, Y., Li, G., Tang, N.: Querying shortest paths on time dependent road networks. PVLDB pp. 1249–1261 (2019)
14. Orda, A., Rom, R.: Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. J. ACM **37**(3), 607–625 (1990)
15. Li, L., Wang, S., Zhou, X.: Time-dependent hop labeling on road network. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 1–12. IEEE (2019)
16. Batz, G.V., Delling, D., Sanders, P., Vetter, C.: Time-dependent contraction hierarchies. In: Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX), p. 97-105 (2009)

17. Li, L., Hua, W., Du, X., Zhou, X.: Minimal on-road time route scheduling on time-dependent graphs. Proc. VLDB Endow. **10**(11), 1274–1285 (2017)
18. Li, L., Zheng, K., Wang, S., Hua, W., Zhou, X.: Go slow to go fast: minimal on-road time route scheduling with parking facilities using historical trajectory. VLDB J. **27**(3), 321–345 (2018)
19. Li, L., Kim, J., Xu, J., Zhou, X.: Time-dependent route scheduling on road networks. SIGSPATIAL Spec. **10**(1), 10–14 (2018)
20. Li, L., Wang, S., Zhou, X.: Fastest path query answering using time-dependent hop-labeling in road network. IEEE Trans. Knowl. Data Eng. (2020)
21. Chen, D., Yuan, Y., Du, W., Cheng, Y., Wang, G.: Online route planning over time-dependent road networks. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 325–335. IEEE (2021)
22. Demiryurek, U., Banaei-Kashani, F., Shahabi, C.: Towards k-nearest neighbor search in time-dependent spatial network databases. In: International Conference on Databases in Networked Information Systems, pp. 296–310 (2010)
23. Komai, Y., Nguyen, D.H., Hara, T., Nishio, S.: Knn search utilizing index of the minimum road travel time in time-dependent road networks. Proceedings of the IEEE Symposium on Reliable Distributed Systems pp. 131–137 (2014)
24. Kolahdouzan, M., Shahabi, C.: Voronoi-based k nearest neighbor search for spatial network databases. In: Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, pp. 84–851. VLDB Endowment (2004)
25. Demiryurek, U., Kashani, F., Shahabi, C.: Efficient k-nearest neighbor search in time-dependent spatial networks. In: Database and Expert Systems Applications, International Conference, Dexa, Bilbao, Spain, August 30-Sept., p. 432-449 (2010)
26. Yang, Y., Li, H., Wang, J., Hu, Q., Wang, X., Leng, M.: A novel index method for k nearest object query over time-dependent road networks. Complexity **2019**, 1–18 (2019)
27. Dreyfus, S.E.: An appraisal of some shortest-path algorithms. Oper. Res. **17**(3), 395–412 (1969)
28. Ouyang, D., Qin, L., Chang, L., Lin, X., Zhang, Y., Zhu, Q.: When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In: SIGMOD, pp. 709–724 (2018)
29. Lee, K.C.K., Lee, W.C., Zheng, B., Tian, Y.: Road: a new spatial object search framework for road networks. IEEE TKDE **24**(3), 547–560 (2012)
30. Abeywickrama, T., Cheema, M.A., Taniar, D.: K-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. VLDB Endowment (2016)
31. Ouyang, D., Wen, D., Qin, L., Chang, L., Lin, X.: Progressive top-k nearest neighbors search in large road networks. In: SIGMOD/PODS '20: International Conference on Management of Data (2020)
32. Papadias, D., Zhang, J., Mamoulis, N., Tao, Y.: Query processing in spatial network databases. In: Proceedings of the 29th International Conference on Very Large Data Bases-Volume 29, pp. 802–813. VLDB Endowment (2003)
33. Akiba, T., Iwata, Y., Kawarabayashi, K., Kawata, Y.: Fast shortest-path distance queries on road networks by pruned highway labeling. Society for Industrial and Applied Mathematics (2014)
34. Demiryurek, U., Banaei-Kashani, F., Shahabi, C., Ranganathan, A.: Online computation of fastest path in time-dependent spatial networks. In: International Symposium on Spatial and Temporal Databases (SSTD), pp. 92–111 (2011)
35. Bidirectional a* search on time-dependent road networks. Networks **59**(2), 240–251 (2012)
36. Foschini, L., Hershberger, J., Suri, S.: On the complexity of time-dependent shortest paths. In: Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 327–341. SIAM (2011)
37. D, D.: Time-dependent sharc-routing. Algorithmica **60**, 60-94 (2011)
38. Zhang, M., Li, L., Zhou, X.: An experimental evaluation and guideline for path finding in weighted dynamic network. Proc. VLDB Endow. **14**(11), 2127–2140 (2021)
39. Zhang, M., Li, L., Hua, W., Zhou, X.: Efficient 2-hop labeling maintenance in dynamic small-world networks. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 133–144. IEEE (2021)
40. Zhang, M., Li, L., Hua, W., Mao, R., Chao, P., Zhou, X.: Dynamic hub labeling for road networks. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 336–347. IEEE (2021)
41. Bodlaender, H.L.: A tourist guide through treewidth acta cybern (1993)
42. Li, M., He, D., Zhou, X.: Efficient knn search with occupation in large-scale on-demand ride-hailing (2020)