Reverse k Nearest Neighbor Search over Trajectories

Sheng Wang, Zhifeng Bao*, J. Shane Culpepper, Timos Sellis, *Fellow, IEEE,* and Gao Cong

Abstract—GPS enables mobile devices to continuously provide new opportunities to improve our daily lives. For example, the data collected in applications created by Uber or Public Transport Authorities can be used to plan transportation routes, estimate capacities, and proactively identify low coverage areas. In this paper, we study a new kind of query – *Reverse k Nearest Neighbor Search over Trajectories* (**RkNNT**), which can be used for route planning and capacity estimation. Given a set of existing routes $\mathcal{D}_{\mathcal{R}}$, a set of passenger transitions $\mathcal{D}_{\mathcal{T}}$, and a query route Q, an **RkNNT** query returns all transitions that take Q as one of its k nearest travel routes. To solve the problem, we first develop an index to handle dynamic trajectory updates, so that the most up-to-date transition data are available for answering an **RkNNT** query. Then we introduce a filter refinement framework for processing **RkNNT** queries using the proposed indexes. Next, we show how to use **RkNNT** to solve the optimal route planning problem **MaxRkNNT** (**MinRkNNT**), which is to search for the optimal route from a start location to an end location that could attract the maximum (or minimum) number of passengers based on a predefined travel distance threshold. Experiments on real datasets demonstrate the efficiency and scalability of our approaches. To the best of our knowledge, this is the first work to study the **RkNNT** problem for route planning.

Index Terms—Trajectory database, route planning, transit network, capacity prediction.

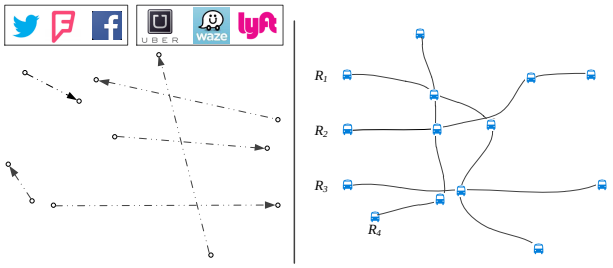
1 Introduction

In the last decade, *Reverse k Nearest Neighbor* (**RkNN**) queries on spatial point data have attracted considerable attention from researchers [5, 7, 23, 24, 29, 31]. An **RkNN** query aims to identify all (spatial) objects that have a query location as a k nearest neighbor. The **RkNN** query has a wide variety of applications such as resource allocation, decision support, and profile-based marketing. For example, **RkNN** queries can be used to estimate the number of customers for planned restaurants among existing restaurants, which is called a *bichromatic* **RkNN**[31].

In addition to point-wise geospatial data, trajectory data describing user movements, such as GPS trajectories of taxis [34] or Uber drivers[1], check-in trajectories [20, 27] of social media users in Foursquare can also provide useful spatial trend data. Recently, such data collected from GPS devices has been used in intelligent transportation applications such as *data-driven passenger flow prediction of bus routes* [1, 9, 13, 15, 26, 33]. The main idea is to find origin-destination data which will help identify a specified route in a large transportation network for commuters. This is essentially a reverse k nearest neighbour search, but the object is a trajectory instead of a single point [24].

In this paper, we will explore **RkNN** search over multiple-point trajectories (referred to as **RkNNT**). In a nutshell, a **RkNNT** query can be described as: taking a planned (or existing) route as a query Q, return all the passengers who will take the query route Q as one of the k nearest routes among the route set $\mathcal{D}_{\mathcal{R}}$. Here, a passenger's movement is modeled as a combination of an origin and a destination [13] such as home and office, which is called a *transition*. Figure 1 presents an example, where there are six *transitions*, each with an origin and destination, which could be collected from social media applications and "Ride Sharing" applications, and four *routes* in a transportation network.

The main difference between our problem **RkNNT** and **RkNN** is that our query is a route, and our data collections contain both routes and user transitions. **RkNNT** can be used to estimate the passengers that will take the query route to travel. Another significant difference between **RkNNT** and **RkNN** is that the transition data is dynamic, and new transitions can arrive continuously, such as requests from Ride-Sharing passengers. Therefore, it is important to take this into consideration in designing a solution when answering an **RkNNT** query.



(a) Origin-destination data (b) Transit network

Fig. 1. Passenger transitions and transit network.

RkNNT queries can serve as a fundamental operation in many applications in the transportation field. The most common one is to estimate the capacity of a route based on passenger movements, as described above. Furthermore, an **RkNNT** query could be used for *Optimal Route Planning* as described below.

Among a set of candidate routes, an **RkNNT** query can be used to find the optimal route which has the maximum (minimum) number of passengers among a set of candidate routes.

* *Corresponding author.*

- *S. Wang, Z. Bao, and J. S. Culpepper are with the School of Science, Computer Science and Information Technology, RMIT University, Australia. Email: {sheng.wang, zhifeng.bao, shane.culpepper}@rmit.edu.au.*
- *T. Sellis is with the Data Science Research Institute, Swinburne University of Technology, Australia. Email: tsellis@swin.edu.au.*
- *G. Cong is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. E-mail: gaocong@ntu.edu.sg.*

Manuscript received xxxx xx, 2017; revised xxxx xx, 2017.

1. https://movement.uber.com

We refer to this problem as a **MaxR*k*NNT** (**MinR*k*NNT**). For Ride-Sharing drivers, finding a route with the maximum number of passengers can increase profitability (the driver fare will be increased with a surge of passenger requests) and the chance of being hired. For ambulance and fire truck drivers, finding a route which has the fewest people around can reduce response time in emergency situations. Furthermore, by taking the temporal factor into consideration (user transitions at different time periods), it can help further estimate the passengers of a bus or car with specific starting and closing hours, in order to save running cost for either individual vehicle drivers or public transportation authority [6].

The main challenge in answering **R*k*NNT** queries lies in how to prune the transitions which cannot be in the results without explicitly accessing every user transition. A straightforward method is to perform a ***k*NN** search for every transition, which is analogous to the *trajectory similarity join problem* [19], and then check the resulting ranked lists to see whether the query is a ***k*NN**. However, this method is intractable when there are a large number of transitions and new transitions are being added to the database.

To overcome the above challenge, we first build two R-tree indexes, the **RR-tree** and **TR-tree**, which are combined with two inverted indexes, **PList** and **NList**, for the route and transition sets, respectively. Then, we choose a set of route points from the existing routes to form a *filtering set* by traversing the **RR-tree**. By drawing bisectors between the route points in the filtering set and the query, an area can be found where any transition point inside cannot have the query as a nearest neighbor. After finding the filtering set, we traverse the **TR-tree** to prune transitions and check if a node can be filtered by more than k routes in the filtering set. Finally, all the candidate transitions are verified using the filtered nodes during the traversal of the **RR-tree**.

Next, we explore the optimal route planning problem-**MaxR*k*NNT** (**MinR*k*NNT**), where we consider a graph formed by a bus network. Given a starting location and a destination, we find the optimal route $\mathcal{R}$ which connects the two locations in a bus network, and maximizes (minimizes) the number of passengers that take $\mathcal{R}$ as its ***k*NN** without exceeding a distance threshold. For **MaxR*k*NNT**, a brute force method can be used to find all candidate routes whose travel distances do not exceed the distance threshold, and then an **R*k*NNT** search can be executed on each candidate, and finally the one with maximum number of passengers is selected as the answer. This method is shown to be inefficient in our experimental study. Similar to **R*k*NNT**, it is crucial to prune candidates which cannot be an optimal route. Another challenge is how to support *dynamic updates* as old transitions expire and new transitions arrive.

For a **MaxR*k*NNT**, a weighted graph is built using the pre-computed **R*k*NNT** set for every vertex. We start from the first vertex and access the neighbor vertex v to compute a partial route R. Then a reachability check on R is performed to see whether the estimated lower bound travel distance of R is greater than the threshold. Next, the dominance table of v is checked to see if R can dominate other partial routes which terminate at v. Further checks on the route are made when R meets all the conditions.

In summary, the main contributions of this paper are:

- We investigate the **R*k*NNT** problem for the first time, which serves as a fundamental yet frequently adopted operator in many practical applications. (See problem definitions in Section 3).
- We propose a filtering-refinement framework which can prune routes using a *filtering set* (in Section 4) and a Voronoi-based optimization to further improve the efficiency (Section 5).
- We introduce **MaxR*k*NNT** (**MinR*k*NNT**) queries which can be used to find the optimal route that attracts the maximum (minimum) number of passengers in a bus network (Section 6).
- We validate the practicality of our approaches using real world datasets (Section 7).

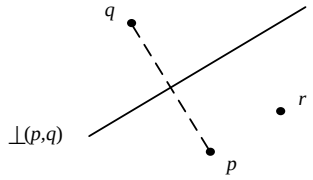


Fig. 2. Half-space pruning.

2 Related Work

In this section, we first compare the difference between our work and classic **R*k*NN** search over point and moving object data. Further, we will review related work on route planning.

2.1 R*k*NN

R*k*NN on Spatial Points. Most existing **R*k*NN** search work focuses on static point data, and often use pruning-refinement frameworks to avoid scanning the entire dataset. However, these approaches cannot easily be translated to route search where both queries and collections consist of multi-point trajectories. Given a set of candidates as a query, a maximizing **R*k*NN** query finds the optimal point with the maximum number of results [35].

Improving search performance has attracted much attention in previous work. The basic intuition behind filtering out a point p is to find another point which is closer to p than the query point q [7, 24, 29]. Here, we review the half space method and use a simple example to show how pruning works. Figure 2 shows a query point q and a data point p. As we can see, a perpendicular bisector divides the whole space into two sub-spaces, and all points inside the lower subspace prefer p as a nearest neighbor, such as point r. For the reverse nearest neighbor search, r may be filtered from the candidate set of query q. More specifically, r can be filtered out if it can be pruned by at least k such points.

R*k*NN on Moving Objects. Given a moving object dataset D and a candidate point set $O = (o_1, o_2, ..., o_m)$ as a query, Shang et al. [21] find the optimal point from O such that the number of moving objects that choose o_i as a nearest neighbor is maximized. Specifically, they proposed a *Reverse Path Nearest Neighbor* (R-PNN) search which finds the nearest point, and not the k nearest points. Shang et al. introduced the concept of *influence-factor* to determine the optimal point. The influence-factor of o is f if o is the nearest neighbor of f trajectories for all candidate points in O.

Cheema et al. [8] proposed a *continuous reverse nearest neighbors* query to monitor a moving object and find all static points that take the moving object as a k nearest neighbor. Recently, Emrich et al. [10] proposed a solution for the problem of RNN search with "uncertain" moving object trajectories using a Markov model approach. A moving object is treated as a result when it always takes the query object as a nearest neighbor within a given time interval. All of these approaches target a single point rather than a transition of multiple-points, which is the focus of our work.

2.2 Route Design and Searching

Traditional Transit Route Design. Bus network design is known to be a complex, non-linear, non-convex, NP-hard problem [9]. Based on existing bus networks, Pattnaik et al. [22] proposed

a heuristic method which uses a genetic algorithm to minimize the cost of passengers and operators. Population estimation and user surveys [30] around the planned route are traditional ways to estimate the number of passengers that may use the planned travel route, such as *Journey to Work*[2] which is data collected as part of a census that describes aspects of commuting behaviour. However, the data is usually out-of-date as census data may only be gathered every five years, and may not reflect current travel patterns.

Data-driven Transit Network Design. A general approach to *transit network design* is to use *demand* data from passengers in addition to *supply* data from existing routes [11]. Supply and demand data is usually abstracted as origin-destination (OD) pairs, and can be used to identify the most profitable routes. With the proliferation of GPS devices on mobile phones and vehicles, demand data can be collected via multiple channels in real time, and several previous studies have exploited such data.

By mining taxi data, Chen et al. [9] tried to approximate night time bus route planning by first clustering all points in taxi trajectories to determine "hot spots" which could be bus stops, and then created bus routes based on the connectivity between two stops. Similarly, based on human mobility patterns extracted from taxi data and smart card data, Liu et al. [13] proposed a localized transportation choice model, which predicts travel demand for different bus routes by taking into account both bus and taxi routes.

Toole et al. [25] used census records and mobile phone location information to estimate demand in transit routes. Historical traffic data [1], smart cards [33], sensors [15], and cellular data [17] can provide more comprehensive demand data and be used to further improve data-driven transit network design. Most of these studies employ data mining methods, which scan the static and historical records for all the customers. In contrast, **R*k*NNT**-based approaches can efficiently maintain up-to-date results whenever there is an update on the trajectory data; moreover, **R*k*NNT** query can be answered efficiently which is critical to real-time decision making, while data mining based approaches cannot. In other words, **R*k*NNT** augments the existing data-driven methods for transit network design.

Optimal Route Searching. Given a starting vertex and an ending vertex, the classical problem is to find the shortest path in a graph. *Best-first Search* (BFS) and *Depth-first Search* (DFS) are two commonly used algorithms for this problem. An extension of this problem is k *Shortest Path searching* (kSP) [2, 32], which aims to find the k shortest paths from a start vertex s to a target vertex t in a directed weighted graph G. Yen's algorithm [32] is a derivative algorithm for ranking the k shortest paths between a pair of nodes. The algorithm always searches the shortest paths in a tree containing the k shortest loop free paths. The shortest one is obtained first, and the second shortest path is explored based on the previous paths. The *Constraint Shortest Path* (CSP) problem [4, 14, 28] applies resource constraints on each edge, and solves the shortest path problem based on these constraints. An example constraint in this scenario is the time cost.

3 Problem Definition

In this section, we formally define the **R*k*NNT** problem and important notations are recorded in Table 1.

2. https://en.wikipedia.org/wiki/Journey_to_work

TABLE 1
Summary of Notation

Notation	Definition
$\mathcal{R}$	The route composed of points $\{r_1, ..., r_n\}$
$\mathcal{T}$	The transition $\{t_o, t_d\}$
Q	Query route $\{q_1, ..., q_m\}$
$\mathcal{D}_\mathcal{R}$, $\mathcal{D}_\mathcal{T}$	The route and transition sets
$dist(t, \mathcal{R})$	Distance from transition point t to $\mathcal{R}$
$\perp(q, r)$	Perpendicular bisector between q and r
$H_{q:r}$, $H_{r:q}$	Two half-planes divided by perpendicular bisector $\perp(q, r)$
$H_{r:Q}$, $H_{\mathcal{R}:Q}$	Filtering space formed by Q with r and $\mathcal{R}$
$\mathcal{C}(r)$	Crossover route set of r
$\mathcal{S}_{filter}$, $\mathcal{S}_{refine}$	Filtering set and filtered node set
$\mathcal{S}_{cnd}$, $\mathcal{S}_{result}$	Transition candidates and result set
$root_r$ ($root_t$)	Root of Route R-tree (Transition R-tree)
$\mathcal{V}_{\mathcal{R},Q}$	Voronoi diagram formed by $\mathcal{R}$ and Q
$\mathcal{G}$	Weighted graph
τ	Travel distance threshold
$\omega(\mathcal{R})$, $\psi(\mathcal{R})$	**R*k*NNT** set and travel distance of $\mathcal{R}$ in $\mathcal{G}$
$\mathcal{M}_\psi[i][j]$	Lower bound matrix of $\psi(\mathcal{R})$ where $\mathcal{R}$ starts from vertex i to j

Definition 1. ***(Route)*** *A route $\mathcal{R}$ of length n is a sequence of points $(r_1, r_2, ..., r_n), n \geq 2$, where r_i is a point represented by (latitude, longitude).*

Definition 2. ***(Transition)*** *A transition $\mathcal{T}$ contains an origin point t_o and a destination point t_d, which is also represented by (latitude, longitude).*

Both a route and a transition are composed of discrete points called *route point* r and *transition point* t, respectively. We use $\mathcal{D}_\mathcal{T}$ and $\mathcal{D}_\mathcal{R}$ to denote the transition set and route set.

Definition 3. ***(Point-Route Distance)*** *Given a transition point $t \in \mathcal{T}$ and a route $\mathcal{R}$, the distance* $\text{dist}(t, \mathcal{R})$ *from t to $\mathcal{R}$ is the minimum Euclidean distance from t to every point of $\mathcal{R}$, and calculated as:*

$$dist(t, \mathcal{R}) = \min_{r \in \mathcal{R}} distance(t, r) \tag{1}$$

Based on the point-route distance function, the ***k*NN** search of a transition point t is defined as:

Definition 4. ***(kNN)*** *Given a set of routes $\mathcal{D}_\mathcal{R}$, the **kNN** search of a transition point $t \in \mathcal{T}$ retrieves a set $S \subset \mathcal{D}_\mathcal{R}$ of k routes such that $\forall \mathcal{R} \in S$, and $\forall \mathcal{R}' \in \mathcal{D}_\mathcal{R} - S$:* $\text{dist}(t, \mathcal{R}) \leq \text{dist}(t, \mathcal{R}')$.

In particular, two types of ***k*NN** are supported for a transition $\mathcal{T}$, which can also be found in [10].

1) $\exists$***k*NN**: $\mathcal{T}$ takes $\mathcal{R}$ as a ***k*NN** iff there exists a point $t \in \mathcal{T}$ taking $\mathcal{R}$ as ***k*NN**. So, $\exists\mathbf{kNN}(\mathcal{T}) = \boldsymbol{k}\mathbf{NN}(t_o) \cup \boldsymbol{k}\mathbf{NN}(t_d)$.
2) $\forall$***k*NN**: $\mathcal{T}$ takes $\mathcal{R}$ as a ***k*NN** iff both points t_o and t_d take $\mathcal{R}$ as their ***k*NN**. So, $\forall\mathbf{kNN}(\mathcal{T}) = \boldsymbol{k}\mathbf{NN}(t_o) \cap \boldsymbol{k}\mathbf{NN}(t_d)$.

Now, we can formally define the reverse k nearest neighbor query over trajectories.

Definition 5. ***(RkNNT)*** *Given a set of routes $\mathcal{D}_\mathcal{R}$, a set of transitions $\mathcal{D}_\mathcal{T}$, and a query route Q, $\exists$**RkNNT**(Q) ($\forall$**RkNNT**(Q)) retrieves all transitions $\mathcal{T} \in \mathcal{D}_\mathcal{T}$, such that $\forall \mathcal{T}$, $Q \in \exists$**kNN**$(\mathcal{T})$ ($\forall$**kNN**$(\mathcal{T})$).*

Example 1. *In Figure 3, R_1, R_2, R_3 and R_4 are* routes. *$T_1, T_2, T_3, T_4, T_5, T_6$ are* transitions*, and T_1^o and T_1^d denote the origin point and destination point for transition T_1. The query route Q is composed of 5 query points (in red). If we take the*

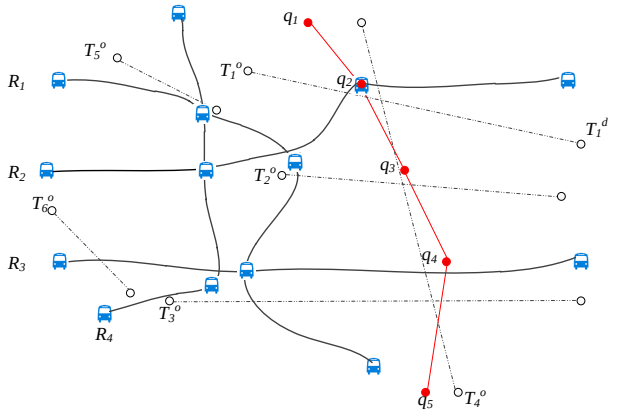


Fig. 3. Example of routes and transitions.

$\forall$***RkNNT*** *query, as point* T_4^o *and* T_4^d *take* Q *as the nearest route,* T_4 *will be the result of* $\forall$***RkNNT***(Q).

Lemma 1. *Given a query* Q, $\forall$***RkNNT***$(Q) \subseteq \exists$***RkNNT***(Q).

Proof. Given a query Q, $\forall$**R*k*NNT**(Q) returns a set of transitions where both origin and destination points have the query as a ***k*NN**, such transitions will also belong to the result of $\exists$**R*k*NNT**(Q), so $\forall$**R*k*NNT**$(Q) \subseteq \exists$**R*k*NNT**(Q). Let $\Delta(Q) = \exists$**R*k*NNT**$(Q) -$ $\forall$**R*k*NNT**(Q), $\forall \mathcal{T} \in \Delta$: $\mathcal{T}$ only has one point that will take the query as a ***k*NN**, so $\Delta(Q) \cap \forall$**R*k*NNT**$(Q) = \emptyset$. □

Using Lemma 1, the set of transition points which take Q as ***k*NN** can be searched for first, and then $\exists$**R*k*NNT**(Q) can be found by adding the corresponding routes. For $\forall$**R*k*NNT**, we need to remove transitions that have only one point in $\exists$**R*k*NNT**(Q). Hence, a unified framework can be proposed that answers both $\exists$**R*k*NNT** and $\forall$**R*k*NNT**. In the rest of this paper, we use **R*k*NNT** to represent $\exists$**R*k*NNT** by default for ease of composition.

4 A Processing Framework for R*k*NNT

In this section, we first provide a sketch of our framework to answer the **R*k*NNT** query for capacity estimation, which includes the pruning idea based on routing points, and the proposed index structures. Then we describe each step in detail.

4.1 Main Idea

Algorithm 1: **R*k*NNT**$(Q, root_r, root_t)$

Input: Q: query, $root_r$: the root node of **RR-tree**, $root_t$: the root node of **TR-tree** (see Section 4.1.2)
Output: $\mathcal{S}_{result}$: the result set.

1 $(\mathcal{S}_{filter}, \mathcal{S}_{refine}) \leftarrow$ FilterRoute$(root_r, Q, k)$;
// Sec 4.2.1
2 $\mathcal{S}_{cnd} \leftarrow$ PruneTransition$(root_t, Q, \mathcal{S}_{filter}, k)$;
// Sec 4.2.2
3 $\mathcal{S}_{result} \leftarrow$ RefineCandidates$(Q, \mathcal{S}_{cnd}, \mathcal{S}_{refine})$;
// Sec 4.2.3
4 **return** $\mathcal{S}_{result}$;

All impossible transitions are pruned using a PruneTransition algorithm, and the remaining candidates $\mathcal{S}_{cnd}$ are further verified using a RefineCandidates algorithm to generate the final result set $\mathcal{S}_{result}$. Before pruning, a subset of routes $\mathcal{S}_{filter}$ needs to be generated for efficient pruning, the reasoning and approach are described in Section 4.1.1. In summary, the whole procedure is composed of the three steps in Algorithm 1.

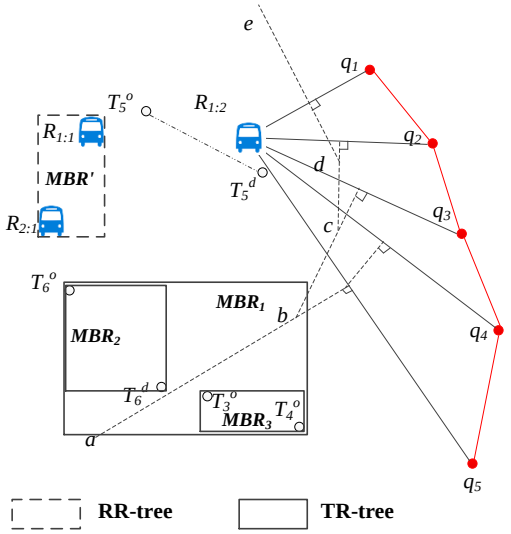


Fig. 4. Half-space pruning for a multi-point query Q.

4.1.1 Pruning Characteristics

By Definition 5, a transition takes a route as a ***k*NN** if there exists at least one point (in the transition) that will take the route as a ***k*NN**. If there are more than k routes which are closer to a point in a transition than the query, then the point in this transition can be pruned. Such a route which helps prune transitions is called a *filtering route*. If both points of a transition are pruned, then the transition can be pruned safely, so the pruning helps to find the filtering routes to prune the transition points.

Lemma 2. *If a transition point* t *is closer to a route point* $r \in \mathcal{R}$ *than* Q*, then* t *is closer to* $\mathcal{R}$ *than* Q.

Proof. We have $dist(t, \mathcal{R}) \leq distance(t, r)$ according to Equation 1. If $distance(t, r) < dist(t, Q)$, then $dist(t, \mathcal{R}) < dist(t, Q)$. □

By Lemma 2, a transition point can be removed if it takes a set of routing points from more than k different routes as a ***k*NN** rather than the query. These points are called *filtering points*. Next, we introduce how to prune a transition point using the filtering point.

Recall the example in Figure 2 where an **R*k*NN** can find an area where the points inside the area will not take the query as the nearest neighbor based on the half space. Similarly, given a query Q, we choose a point r from a route $\mathcal{R}$ in $\mathcal{D}_{\mathcal{R}}$; then based on the straight line $\overline{rq_i}$ formed by a point q_i in Q to r, the perpendicular bisector $\perp(q_i, r)$ is used to cut the space into two half-planes: $H_{r:q_i}$ and $H_{q_i:r}$ which contain r and q_i, respectively. For every point q_i in Q, there is a $H_{r:q_i}$. The intersection of all of the half spaces forms the filtering space $H_{r:Q}$ defined as:

Definition 6. ***(Filtering Space)*** *Given a route point* r *and a query* Q*, the intersection of all* $H_{r:q_i}$ *forms a filtering space:*

$$H_{r:Q} = \bigcap_{q_i \in Q} H_{r:q_i} \tag{2}$$

point r *which belongs to* $\mathcal{R} \subset \mathcal{D}_{\mathcal{R}}$ *is called as the filtering point.*

As shown in Figure 4, we can see that there are five perpendicular bisectors. They form a polyline $\overline{abcde}$ which divides the whole space into two sub-spaces, and the left part is the filtering space $H_{R_{1:2}:Q}$. As T_5 is entirely located in this area, it cannot take the query as its nearest route. The filtering space can also help filter a set of points using spatial indexes (see Sec. 4.1.2).

If a maximum bounded rectangle (MBR) such as MBR_2 covering points T_6^o and T_6^d is located entirely inside the filtering space, then T_6^o and T_6^d inside this MBR will not take the query as a nearest neighbor and can be filtered out.

Every point in the route set $\mathcal{D}_\mathcal{R}$ can be a filtering point, but we cannot choose all points in the route set $\mathcal{D}_\mathcal{R}$ to do pruning especially when the whole set is large and located in external memory. When pruning a transition point, the process is costly if we access all route points every time. In Section 4.2.1, we introduce how to generate a subset from the whole route set. Overall, we can observe three key characteristics based on the above analysis: 1) A filtering space exists between the query and a route point; 2) If a transition point is located in more than k filtering space of query Q simultaneously, then the point can be pruned; and 3) It is important to choose a subset of all routes as the filtering set.

4.1.2 Indexes

- **RR-tree** & **TR-tree** are two tree indexes for point data fetched from route dataset $\mathcal{D}_\mathcal{R}$ and transition dataset $\mathcal{D}_\mathcal{T}$, respectively, and referred to as a *Route R-tree* (**RR-tree**) and a *Transition R-tree* (**TR-tree**). The tree indexes are created first, and every point in the leaf node of **RR-tree** contains the ID for its route. Every point in **TR-tree** also contains the IDs of the transition it belongs to. Through the transition ID and route ID in the node of **RR-tree** and **TR-tree**, we can get the corresponding route and transition for further refinement if two points of a transition are both pruned, and the two filtering points belong to the same route (See Section 4.2.3).
- **NList**. We need to find all of the routes that have a point inside a given node for verification. Hence, for each node in **RR-tree**, we create a list for every node of **RR-tree** by traversing the whole tree bottom-up to store all of the IDs for routes inside.
- **PList**. The inverted list of each route point is created to store the IDs of the corresponding routes. As a bus stop can be shared by many routes in a bus network, we call this index a **PList**.

Our index supports dynamic updates, where new transitions and routes can be added into the index easily.

4.2 Key Functions

This section describes: 1) how to generate the filtering set $\mathcal{S}_{filter}$; 2) how to prune and find all the candidate routes $\mathcal{S}_{cnd}$; and 3) how to verify the candidate routes and further refine them to find the final query result $\mathcal{S}_{result}$.

4.2.1 Filtering Routes

In order to get a small filtering set $\mathcal{S}_{filter}$ for a given query, an empty filtering set $\mathcal{S}_{filter}$ is initialized, and new route points are added which cannot be pruned using the existing points of a route in $\mathcal{S}_{filter}$. We organize all of the filter points into a point list, sorted by the number of routes which cover each point, and denote the route set and point set as $\mathcal{S}_{filter}.R$ and $\mathcal{S}_{filter}.P$, respectively, which are materialized using two dynamically sorted hashtables. Specifically, for $\mathcal{S}_{filter}.R$, the key is the route ID, and the values are points of this route that cannot be filtered. For $\mathcal{S}_{filter}.P$, the key is the route point ID, and the value is a list of routes containing the point.

In a real bus network, a route point can be covered by several routes. If a filtering point is contained by more than k routes, and a transition takes this filtering point as the nearest neighbor rather than the query, then this transition point can be pruned. We will employ this enhancement to achieve more efficient pruning.

Definition 7. ***(Crossover Route Set)*** *Given a route point r, the set of routes which cover r is r's crossover set, and denoted as $\mathcal{C}(r)$.*

For example in Figure 3, R_1 and R_4 intersect at the second point $R_{1:2}$, then $\mathcal{C}(R_{1:2}) = \{R_1, R_4\}$. Using the **PList**, we can retrieve the *crossover route set* of a point r easily, where $\mathcal{C}(r) = PList[r]$. The crossover route set of each filter point $r \in \mathcal{S}_{filter}.P$ can be sorted by $|\mathcal{C}(r)|$ to give higher priority to the points which are crossed by more routes in the filtering phase.

Starting from the root node of **RR-tree**, the filtering algorithm iteratively accesses the entries of **RR-tree** from a heap in ascending order of their minimum distances to the query Q. The accessed points are used for filtering the search space. If an accessed entry e of index can be filtered – e is pruned by more than k routes – it can be skipped (see Algorithm 3). Otherwise, if e is an intermediate or leaf node, its children are inserted into the heap; if e is a route point and cannot be filtered, it is inserted in the filter set $\mathcal{S}_{filter}$ and its half-space is used to filter the search space. The filtering algorithm terminates when the heap is empty. The details can be found in Algorithm 2.

Algorithm 2: FilterRoute($root_r$, Q, k)

```
Output: S_filter: filtering set, S_refine: filtered node set for
        refinement
1  minheap ← ∅, S_filter ← ∅, ← ∅;
2  minheap.push(root_r);
3  while minheap.isNotEmpty() do
4      e ← minheap.pop();
5      if e is a node then
6          if isFiltered(Q, S_filter, e, k) then
7              S_refine.add(e);        // Filtered node set
8              continue;
9      if e is a point then
10         S_filter.add(e, C(e));          // Filtering set
11     else
12         foreach child c of e do
13             minheap.push(c, MinDist(Q, c));
14 return (S_filter, S_refine);
```

The minimum distance from a child c to the query is computed as the minimum distance from every query point to the node c:

$$MinDist(Q, c) = \min_{q \in Q} MinDist(q, c) \tag{3}$$

In Line 10 of Algorithm 2, a point that cannot be pruned is a filter point and is added into $\mathcal{S}_{filter}$. First the route ID of the point is found, and inserted into $\mathcal{S}_{filter}.R$. Then the point is inserted into the corresponding sorted point list $\mathcal{S}_{filter}.P$, and each point is affiliated with a list of route IDs containing it.

Algorithm 3 shows how the filtering works. The filtering of a node is conducted in two steps. In step 1 (Line 2-2), the filter points $\mathcal{S}_{filter}.P$ are processed to do the filtering. All points in $\mathcal{S}_{filter}.P$ are sorted by the size of their crossover route set, and each point is accessed in descending order. If a filtering point is found that can filter the node, then all affiliated route IDs are added to S. If S contains more than k unique route IDs, termination occurs and the node can be filtered out. After checking all the filtering points, step 2 (Line 11-15) is initiated, and the routes inside $\mathcal{S}_{filter}.R$ are used for filtering. Finally, the filtering method based on Voronoi diagrams is employed (Section 5.1).

Algorithm 3: IsFiltered(Q, $\mathcal{S}_{filter}$, *node*, k)

Output: whether the *node* can be filtered
$S \leftarrow \emptyset$;
foreach $p \in \mathcal{S}_{\text{filter}}.P$ **do**
  // access list points in descending order
  if $S.size > k$ **then**
    return *true*;
  label $\leftarrow$ *true*;
  foreach $q \in Q$ **do**
    if *node located in* $H_{p:q}$ **then**
      label $\leftarrow$ *false*;
  if *label* $=$ *true* **then**
    $S \leftarrow S \cup \mathcal{C}(p)$; // crossover route set
$S' \leftarrow \mathcal{S}_{filter}.R - S$;
foreach *route* $\in S'$ **do**
  if $S.size > k$ **then**
    return *true*;
  // see Section 5.1
  if VoronoiFiltering(Q, *route*, *node*) **then**
    $S \leftarrow S \cup \{route\}$;
return *false*;

Example 2. *Recall the query in Figure 4. We access the* ***RR-tree*** *to form the filtering point set from root to leaf. Initially, the filtering set is composed of the points which are closest to the query, such as* $R_{1:2}$. *Given an* MBR$'$ *that bounds* $R_{2:1}$ *and* $R_{1:1}$, *Algorithm 3 can be used to prune* MBR$'$, *as it is completely located inside the filter space, but must still be added to* $\mathcal{S}_{\text{refine}}$ *for further verification as discussed in Section 4.2.3.*

4.2.2 Transition Pruning

Based on the filter set $\mathcal{S}_{filter}$, entries e from **TR-tree** are added to a heap which is sorted by the distance to the query in ascending order, and checked to see if they can be pruned by $\mathcal{S}_{filter}$ using Algorithm 3. Algorithm 3 uses the candidates in $\mathcal{S}_{filter}$ to check whether e is located in a filtering space of Q. The transition points that cannot be pruned are considered as candidate for refinement.

Algorithm 4: PruneTransition($root_t$, Q, $\mathcal{S}_{filter}$, k)

Output: $\mathcal{S}_{cnd}$: candidate set
minheap $\leftarrow \varnothing$, $\mathcal{S}_{cnd} \leftarrow \varnothing$;
minheap.push($root_t$);
while *!minheap.isEmpty()* **do**
  $e \leftarrow$ *minheap.pop()*;
  if *e is a Node* **then**
    if isFiltered*(Q,* $\mathcal{S}_{\text{filter}}$*, e,* k*)* **then**
      continue;
  if *e is a point* **then**
    $\mathcal{S}_{cnd}.add(e)$;
  else
    foreach *child c of e* **do**
      minheap.push(c, *MinDist*(c, Q));
return $\mathcal{S}_{cnd}$;

Algorithm 4 describes the procedure to prune the transition points using the generated filter set $\mathcal{S}_{filter}$ from **TR-tree**. It is similar to the filtering method for generating the filtering set. The main difference with the traversal of **RR-tree** is that only the unpruned points need to be stored, and the filtering set $\mathcal{S}_{filter}$ is fixed. As a result, a set of transition points $\mathcal{S}_{cnd}$ is obtained which takes the query routes as k nearest neighbors.

Example 3. *To prune the transitions, each node is checked with* ***TR-tree*** *to see if it can be pruned by* $\mathcal{S}_{\text{filter}}$. *As shown in Figure 4, when* $R_{1:2}$ *is used to prune* MBR_1 *with Algorithm 3, it cannot be pruned as it is not located completely within the filtering space* $H_{R_{1:2}} : Q$. *If it is not pruned by other points in the filtering route set in the end, its children* MBR_2 *and* MBR_3 *will be accessed, and further pruning of* MBR_2 *is required as it is located in the filtering space. For* MBR_3, *if it is not pruned by any route point, then its children* T_3^o *and* T_4^o *are added into the candidate set* $\mathcal{S}_{\text{cnd}}$.

4.2.3 Candidate Verification

Candidate verification is implemented in the RefineCandidates method. It mainly uses the filtered node set $\mathcal{S}_{refine}$ during the traversal of **RR-tree** to find $\mathcal{S}_{filter}$ in Algorithm 2. It can be divided into two steps. First, the nodes in **RR-tree** encountered during the filtering phase are kept in $\mathcal{S}_{refine}$ in Line 7 of Algorithm 2. The verification algorithm runs in rounds. In each round, one of the nodes in $\mathcal{S}_{refine}$ is opened and its children are inserted into $\mathcal{S}_{refine}$. During each round, the nodes and points in $\mathcal{S}_{refine}$ are used to identify the candidates that can be verified using $\mathcal{S}_{refine}$, which are the nodes confirmed as **R*k*NNT**, or guaranteed not to be **R*k*NNT**. Such candidates are verified and removed from $\mathcal{S}_{cnd}$. The algorithm terminates when $\mathcal{S}_{cnd}$ is empty. The result set is then stored for a second round of verification.

To verify a candidate effectively, if more than k routes are found in $\mathcal{S}_{refine}$ which are closer to the query than the candidates, then it can safely be removed from $\mathcal{S}_{cnd}$. Hence, we maintain a set to store the unique IDs of these routes when every candidate point is checked in $\mathcal{S}_{cnd}$. The route IDs are found, and the set is updated using **NList** when new filtering points or nodes from $\mathcal{S}_{refine}$ are found. When the number of IDs in a set is greater than k, it can be removed from $\mathcal{S}_{cnd}$.

After finding the transition points for the routes, they will take the query as k nearest neighbors, so for the ∃**R*k*NNT**, the transition ID for all remaining points can be returned as the final result $\mathcal{S}_{result}$ in the second step. For ∀**R*k*NNT**, if a transition only has one point in the result set, then it will be pruned, and only the transitions which have both points in the result will be considered as the real result and added to $\mathcal{S}_{result}$.

Example 4. *In the verification stage,* $\mathcal{S}_{\text{refine}}$ *is used to verify* $\mathcal{S}_{\text{cnd}}$ *because* $\mathcal{S}_{\text{filter}}$ *cannot prune* $\mathcal{S}_{\text{cnd}}$. *For* T_3^o *and* T_4^o *in* $\mathcal{S}_{\text{cnd}}$, MBR$'$, *which covers* $R_{2:1}$ *and* $R_{1:1}$, *is chosen to compute the minimum distance. If the node is closer to the transition point, such as is the case with* T_3^o, *the candidate can be pruned as it will not choose the query as nearest neighbor; otherwise, its child nodes are accessed and checked until there are more than* k *routes closer to the candidate than the query route. After checking all the nodes in* $\mathcal{S}_{\text{refine}}$, *the surviving points in* $\mathcal{S}_{\text{cnd}}$ *are returned as the result.*

4.3 Computational Complexity Analysis

In this section, we will analyze the time complexity of Algorithm 1. We have two datasets: $\mathcal{D}_{\mathcal{R}}$ and $\mathcal{D}_{\mathcal{T}}$, here we use $|\mathcal{D}_{\mathcal{R}}|$ and $|\mathcal{D}_{\mathcal{T}}|$ to denote the number of points in each dataset. Let the number of filtering points be $|\mathcal{S}_{filter}|$ and the number of filtered route nodes be $|\mathcal{S}_{refine}|$.

First, we analyze the complexity of filtering a node. Every filtering point is checked against nodes or points, and costs

$\mathcal{O}(k \cdot |\mathcal{S}_{filter}| \cdot |Q|)$ at most. The filtering complexity over **RR-tree** is $\mathcal{O}(k \cdot |\mathcal{S}_{filter}| \cdot |Q| \cdot |N_{MBR}(\textbf{RR-tree})|)$, where $|N_{MBR}|$ is the number of scanned nodes in **RR-tree**. Similarly, the complexity of filtering and verification over **TR-tree** will be $\mathcal{O}(k \cdot |\mathcal{S}_{filter}| \cdot |Q| \cdot |N_{MBR}(\textbf{TR-tree})|)$ and $\mathcal{O}(k \cdot |\mathcal{S}_{refine}| \cdot |Q| \cdot |N_{MBR}(\textbf{TR-tree})|)$, respectively. Hence, the total complexity is $\mathcal{O}(k \cdot \frac{|\mathcal{D}_\mathcal{T}|}{f} \cdot |\mathcal{S}_{filter}| \cdot |Q|) + \mathcal{O}(k \cdot \frac{|\mathcal{D}_\mathcal{T}|}{f} \cdot |\mathcal{S}_{refine}| \cdot |Q|)$, as the number of scanned nodes are proportional to the number of facilities and routes, where f is the fanout of the R-tree. In most cases, $|\mathcal{D}_\mathcal{T}| \gg |\mathcal{D}_\mathcal{R}|$, so we can ignore the route filtering in Algorithm 2, and the complexity is:

$$\mathcal{C}(\textbf{Rk}\textbf{NNT}) = \mathcal{O}(k \cdot \frac{|\mathcal{D}_\mathcal{T}|}{f} \cdot (|\mathcal{S}_{filter}| + |\mathcal{S}_{refine}|)| \cdot |Q|) = \mathcal{O}(k \cdot \frac{|\mathcal{D}_\mathcal{T}| \cdot |\mathcal{D}_\mathcal{R}|}{f^2} \cdot |Q|) \quad (4)$$

Note that the fanout f is usually set according to the page size of disk to reduce I/O costs [12, 31]. A large f results in a case that leaf nodes cannot be completely located in the filtering space, so the probability of it being filtered is reduced, and its children must be scanned too, which leads to increased I/O costs.

5 Optimization of the Filtering Process

5.1 Voronoi-based Filtering

One problem of the filtering method in Algorithm 3 is that the filtering space obtained from a single point and the query is usually very small. For example in Figure 4, MBR_1 cannot be pruned, so it needs to load MBR_2 and MBR_3 to perform further checks, which require additional pruning time. To further enlarge the pruning space, the available filtering points in a single route can be used rather than a single point to perform the pruning, namely, $\mathcal{S}_{filter}$ can be used for additional pruning. Given a query and a filtering route $\mathcal{R}$, a larger filter space can be explored. To find this area, Voronoi cells can be used.

To accomplish this, a plane is partitioned with points into several convex polygons, such that each polygon contains exactly one generating point, and every point in a given polygon is closer to its generating point than to any other. The convex polygon of one point is the *Voronoi cell*, and the point is the *kernel* of this cell. By plotting the Voronoi diagram $\mathcal{V}_{\mathcal{R},Q}$ between the query Q and a filtering route $\mathcal{R}$, as shown in Figure 5, the Voronoi cell $\mathcal{V}_{\mathcal{R},Q}[p]$ of the route $\mathcal{R}$ can be found, and any point inside these cells are closer to the filtering route than the query. Furthermore, if a node does not intersect with any cell of the query, then any point inside this node will be closer to the filtering route than the query. If a node can find more than k such filtering routes, then the node can be pruned.

Definition 8. *(**Voronoi Filtering Space**) Given a filtering route $\mathcal{R}$ and query Q, we define the Voronoi filtering space as:*

$$H_{\mathcal{R}:Q} = \bigcup_{p \in \mathcal{R}} \mathcal{V}_{\mathcal{R},Q}[p] \quad (5)$$

which is the union of the Voronoi cells of all points from $\mathcal{R}$, and $\mathcal{V}_{\mathcal{R},Q}$ is the Voronoi diagram of the union of the points from $\mathcal{R}$ and Q.

Any transition point inside $H_{\mathcal{R}:Q}$ cannot have Q as the nearest neighbor. As shown in Figure 5, for any point in the Voronoi filtering space, it can find a point in the filtering route which is closer than any point in the query. Hence, two points in a transition

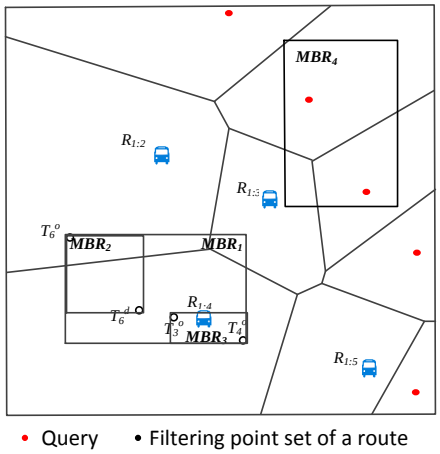


Fig. 5. Pruning based on the Voronoi diagram of a query (red) and a filtering route composed of 4 points (black).

can both find a point in the filtering route rather than the query, so the transition point will not take the query as a nearest neighbor.

The filtering route $\mathcal{R}$ is used to further prune the transition point if it cannot be pruned by the filter points in $\mathcal{R}$ one by one as shown in Line 2-9 of Algorithm 3. After scanning all the filtering points of a route in $\mathcal{S}_{filter}.P$, we will use the Voronoi filtering space of the route for the query to prune the transition points, where the space has been created after getting the filtering route set. Then the pruning space will be larger, and we can find another route which is closer to the entry than the query if it can prune the entry.

For example, consider the 4 points belonging to a same route R_1 to prune the transition points in Figure 5. The filtering space is larger than the area shown in Figure 4, and MBR_1 is entirely located within the filtering space, so it can be pruned from consideration. Since the Voronoi diagram can be produced at the same time as when the perpendicular bisectors from query to every filtering point are computed, then there is no additional cost to generate the Voronoi information. This additional pruning rule improves the probability of a node being pruned.

5.2 Divide & Conquer Method

Note that the processing costs of the proposed method is high when the query has many points. The main reason is that a node has to be filtered by every query point, and the probability of a point being pruned will be lower when the query is long. To alleviate this problem, we introduce a divide-and-conquer method based on our processing framework.

Lemma 3. *The **RkNNT** of a multi-point query is the union of the **RkNNT** of all points in a query:*

$$\textbf{RkNNT}(Q) = \bigcup_{q_i \in Q} \textbf{RkNNT}(q_i) \quad (6)$$

Proof. For a transition, if it takes a query point as a k nearest neighbor, then it must be a **RkNNT** result for Q, so $\bigcup_{q_i \in Q} \textbf{RkNNT}(q_i) \subseteq \textbf{RkNNT}(Q)$. For each transition in $\textbf{RkNNT}(Q)$, it must take one query point in Q as the **kNN**, then $\textbf{RkNNT}(Q) \subseteq \bigcup_{q_i \in Q} \textbf{RkNNT}(q_i)$. Based on these two observations, $\textbf{RkNNT}(Q) = \bigcup_{q_i \in Q} \textbf{RkNNT}(q_i)$. □

Based on this observation, a divide and conquer framework is proposed that uses multiple **RkNNT** searches which were introduced in Section 4. The main idea is that **RkNNT** search is performed for every query point to find a candidate transition point

set for every query point first, and then the transitions containing these points are merged to get the final transition result.

Even though an **R*k*NNT** query mainly targets a route query, it can process single-point queries as well since every step in the algorithm does not require that the query have more than one point. According to Definition 6, the filtering space will be the largest when there is only one query point, so the pruning efficiency will be the highest when compared with any multi-point query which extends from this single query point.

6 Optimal Route Planning

In this section, we present a solution to the route planning problem with a distance threshold based on **R*k*NNT**. We first define a new query called **MaxR*k*NNT**. A baseline method is proposed first, and then an efficient method based on pre-computation and pruning is described.

6.1 Maximizing R*k*NNT in a Bus Network

In bus route planning, the goal is to attract the maximum number of passengers within a given distance threshold, since a single bus cannot cover all stops in a city. For Uber drivers, such a route also means more opportunities to maximize profit by attracting bus passengers to use alternative modes of transportation. Next, we will introduce the maximizing **R*k*NNT** for bus networks.

Here we use the real bus networks in NYC and LA as an example. Figure 6 shows that the ratio between the travel distance and the straight line distance between start and end bus stops, and does not exceed 2 in most bus routes. Hence, such a distance constraint always exists in real-life route planning.

We first cast the existing bus network as a *Weighted Graph*.

Definition 9. ***(Weighted Graph)*** *$\mathcal{G} = (E, V)$ is a weighted graph, where V is the vertex set and E is a set of edges which connect two vertices among V. A route in $\mathcal{G}$ is a sequence of vertices $R = (v_1, v_2, \ldots, v_n) \in V \times V \times \cdots \times V$ such that v_i is adjacent to v_{i+1} for $1 \leq i < n$, v_1 and v_n are the start and end vertex.*

Given a route $\mathcal{R}$, $\psi(\mathcal{R})$ is the travel distance starting from start to end through every vertex in the route:

$$\psi(\mathcal{R}) = \sum_{p_i \in \mathcal{R} \& i \in [1, n-1]} distance(p_i, p_{i+1}) \tag{7}$$

Recall Definition 5, given a route $\mathcal{R}$ in $\mathcal{G}$, among the transition set $\mathcal{D}_\mathcal{T}$, the **R*k*NNT** of $\mathcal{R}$ can find all transitions that would choose it as a ***k*NN**. The passengers who are likely to take $\mathcal{R}$ are the **R*k*NNT** set of $\mathcal{R}$. Let $\omega(\mathcal{R}) = \textbf{R}k\textbf{NNT}(\mathcal{R})$ for simplicity. We now formally define the Maximizing **R*k*NNT** (**MaxR*k*NNT**) problem for route planning.

Definition 10. ***(MaxRkNNT)*** *Given a threshold τ, a starting vertex v_s and a destination vertex v_e, **MaxRkNNT**(v_s, v_e, τ) returns an optimal route R from S_{se} such that $\forall R^{'} \in S_{se} - R$, $|\omega(R)| \geq |\omega(R^{'})|$ and $\psi(R) \leq \tau$, where S_{se} is the set of all possible routes in $\mathcal{G}$ that share same start and end vertex.*

The definition of **MinR*k*NNT** can be defined by changing $|\omega(R)| \geq |\omega(R^{'})|$ to $|\omega(R)| \leq |\omega(R^{'})|$ in Definition. 10. In this paper, we propose a search algorithm which can solve both **MaxR*k*NNT** and **MinR*k*NNT**. By default, we choose **MaxR*k*NNT** for ease of illustration.

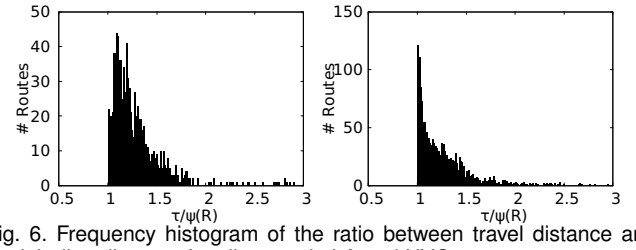


Fig. 6. Frequency histogram of the ratio between travel distance and straight-line distance for all routes in LA and NYC.

Baseline. The simplest brute force method for **MaxR*k*NNT** is to find all candidate routes which meet the travel distance threshold constraint. This can be accomplished by extending the *k shortest path* method proposed by Yen [32] and also by Martins et al. [16] with a loop to find the sub-optimal route until the distance threshold τ is met. Then a **R*k*NNT** query is ran for each candidate and the one with maximum number of results as the optimal route is selected. We call this method **BF**. Recall the query in Figure 7, where almost all routes such as $\overline{abej}$, $\overline{acej}$ and $\overline{acehj}$ will be candidates.

However, the performance of **R*k*NNT** decreases as the number of points increases, which is discussed in more detail in Section 7.3. For bus route planning, it may be tolerable to wait for a few seconds to conduct **MaxR*k*NNT** query. However, for real time queries, like identifying profitable routes for Uber drivers, this method is less desirable. To achieve better performance, an efficient route searching algorithm is proposed based on the pre-computation of the **R*k*NNT** set for each vertex in $\mathcal{G}$.

Complexity of baseline method. The complexity of state-of-the-art approach for k shortest path (**kSP**) search [2] is

$$\mathcal{C}(\textbf{kSP}) = \mathcal{O}(|E| + |V|log|V| + |R|) \tag{8}$$

where $|R|$ is the number of routes that do not exceed the travel distance threshold from source to end in the graph. To answer a **MaxR*k*NNT**, we need to know the **R*k*NNT** of each scanned vertex during graph traversal. So, the overall time complexity can be computed as

$$\mathcal{C}(\textbf{MaxR}k\textbf{NNT}) = \mathcal{C}(\textbf{R}k\textbf{NNT}) \cdot |R| + \mathcal{C}(\textbf{kSP}) = \mathcal{O}(k\frac{|\mathcal{D}_\mathcal{T}||\mathcal{D}_\mathcal{R}|}{f^2}|Q||R|) + \mathcal{O}(|E| + |V|log|V| + |R|) \tag{9}$$

As we can see, it is impossible to search for the $|R|$ shortest paths in real time for a graph containing a large amount of transition data. Moreover, the number of candidate routes from **kSP**. $|R|$ is also very large when the constraint of travel distance is loosely interpreted. To avoid spending too much time on an **R*k*NNT** for a vertex, a better solution is to pre-compute them and update the capacity of each vertex regularly when inserting new transitions. Moreover, using pre-computed capacities can help filter out impossible routes in advance. More details on this idea can be found in Section 6.2.2.

6.2 Our Solution

According to Lemma 3, the query Q can be decomposed into $|Q|$ points, which means that we can get the pre-computed **R*k*NNT** set for every vertex, and perform a union operation on all vertices in a route to get the final **R*k*NNT** set for that route.

By using the above property, we introduce a pre-computation based method with a fixed k which provides better performance. Note that even though k should be fixed in the pre-computation,

multiple datasets of representative k can be generated in advance to meet different requirements.

6.2.1 Pre-computation

For every vertex in $\mathcal{G}$, an **R*k*NNT** query is ran, and the result stored. A pre-computed matrix $\mathcal{M}_\psi[i][j]$ is created which stores the pre-computed all-pair shortest distance for all vertexes in $\mathcal{G}$ using the Floyd-Warshall algorithm [18]. The details of pre-computation can be found in Algorithm 5.

Algorithm 5: Precomputation($\mathcal{G}$, $\mathcal{D}_\mathcal{R}$, $\mathcal{D}_\mathcal{T}$, k)

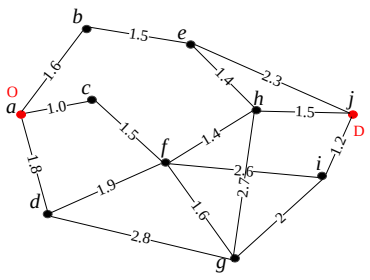
Output: $\mathcal{G}.V$: the vertexes with **R*k*NNT** set
$root_t \leftarrow CreateIndex(\mathcal{D}_\mathcal{T})$; // root of **TR-tree**
$root_r \leftarrow CreateIndex(\mathcal{D}_\mathcal{R})$; // root of **RR-tree**
foreach *vertex* $v \in \mathcal{G}$ **do**
  $\mathcal{S}_{result} \leftarrow$ **R*k*NNT**$(v, root_r, root_t)$; // call Algorithm 1 by query v
  $\mathcal{G}.V.$**R*k*NNT**$(v) \leftarrow \mathcal{S}_{result}$; // update the set on vertex
  foreach *vertex* $v' \in \mathcal{G} - v$ **do**
    $\mathcal{M}_\psi[v][v'] \leftarrow$ ShortestDistance$(\mathcal{G}, v, v')$;
return $\mathcal{G}.V$;

With the pre-computed **R*k*NNT** set, we can further improve the performance of the baseline method **BF**. After getting all candidate routes that do not exceed the distance threshold, the **R*k*NNT** set of each route can be found by performing a union operation on the sets. Compared with the baseline method, the on-the-fly **R*k*NNT** query is replaced with pre-computation, and the running time is reduced to the search time of k shortest path search. However, it is still possible to leverage distance constraints and dominance relationships to prune additional routes in advance.

Example 5. *As shown in Figure 7, the red points are the start $O = a$ and end $D = j$ respectively. A query formed by these two points and $\tau = 6$ return the route with largest **RkNNT** set, where the number on each edge is the distance between two vertices, and the label is the vertex ID. The table shows the pre-computed **RkNNT** set for each vertex. So, $\omega(\overline{acfhj}) = \{T_1, T_2, T_3, T_4, T_6\}$ and $\psi(\overline{acfhj}) = 1 + 1.5 + 1.4 + 1.5 = 5.4$.*

6.2.2 Route Searching by Pruning

After getting the **R*k*NNT** set for every vertex in the graph $\mathcal{G}$, Algorithm 6 can be ran to get the optimal route based on the pre-computed Euclidean distance of every edge. Specifically, the neighbor vertices are accessed around the starting point, and two levels of checking are performed to see whether the current partial route R^* is feasible. If it is, it is inserted into the priority heap $\mathcal{Q}$, and the partial route is increased until it meets the end point v_e and has the maximum result set size.



Vertex	**R*k*NNT** Set
a	T_1^o
b	T_1^d
c	T_1^d, T_3^o, T_4^o
d	T_5^o
e	T_2^o
f	T_2^o, T_3^d, T_4^d
g	T_5^o
h	T_2^d
i	T_6^o
j	T_6^d

Fig. 7. An exemplar graph with a query $(a, j, 6)$ where a and j are the start and end vertexes, $\tau = 6$ is the distance threshold, and the table shows the **R*k*NNT** set for each vertex.

Algorithm 6: MaxR*k*NNT(o, d, τ)

Output: R: the optimal route
if checkReachability(v_s, v_e, τ) **then**
  return $\varnothing$;
$R \leftarrow \varnothing$, $R^* \leftarrow \{v_s\}$;
$\psi(R^*) \leftarrow 0$; // travel distance
$\omega(R^*) \leftarrow \mathcal{G}.V.$**R*k*NNT**$(v_s)$; // **R*k*NNT** set
$\mathcal{Q} \leftarrow \varnothing$; // queue stores the partial routes
$push(\mathcal{Q}, \{R^*, \psi(R^*), \omega(R^*)\})$;
$max \leftarrow |\omega(R^*)|$;
while $\mathcal{Q} \neq \varnothing$ **do**
  $\{R^*, \psi(R^*), \omega(R^*)\} \leftarrow pop(\mathcal{Q})$;
  $v_i \leftarrow GetEnd(R^*)$;
  foreach $v_j \in Neighbor(\mathcal{G}, v_i)$ **do**
    if checkReachability$(v_j, d, \tau - \psi(R^*))$ **then**
      if checkDominance$(o, v_j, \psi(R^*), \omega(R^*))$ **then**
        $S \leftarrow Update(\mathcal{DT}[d], \psi(R^*), \omega(R^*)))$;
        foreach *candidate* $\in S$ **do**
          $Delete(\mathcal{Q}, candidate)$;
        $R^* \leftarrow R^* \cup \{v_j\}$;
        $\psi(R^*) \leftarrow \psi(R^*) + \psi(v_i, v_j)$;
        $\omega(R^*) \leftarrow \omega(R^*) \bigcup \mathcal{G}.V.$**R*k*NNT**$(v_j)$;
        $push(\mathcal{Q}, \{R^*, \psi(R^*), \omega(R^*)\})$;
  if $GetEnd(R^*) = v_e$ **then**
    if $|\omega(R^*)| >$ max **then**
      // new optimal route
      $R \leftarrow R^*$;
      $max \leftarrow |\omega(R^*)|$;
return R;

The two checking functions are:

1) checkReachability: This pruning function checks whether the current route meets the distance constraint – namely that the distance from the current vertex to the end vertex is less than $\tau - \psi(R^*)$. When $\mathcal{M}_\psi[v_j][d] > \tau - \psi(R^*)$, it will return false and move to the next neighbor of vertex v_i in $\mathcal{G}$.
2) checkDominance: This pruning function exploits the dominance relationship between two partial routes. If a partial route exists that ends at the same vertex and has a shorter route and a larger **R*k*NNT** set, then it can dominate the current route.

The following dominating lemma is introduced which works for both ∀**R*k*NNT** and ∃**R*k*NNT**.

Lemma 4. *Given two partial routes R_1^* and R_2^* which have the same start and end, R_1^* dominates R_2^* in **MaxRkNNT** (R_2^* dominates R_1^* in **MinRkNNT**) when $|\psi(R_1^*)| < |\psi(R_2^*)|$ and $|\forall$**RkNNT**$(R_1^*)| > |\exists$**RkNNT**$(R_2^*)|$.*

Proof. To prove the lemma, we use $\omega(\mathcal{R})$ and $\omega^*(\mathcal{R})$ to represent ∃**R*k*NNT**$(\mathcal{R})$ and ∀**R*k*NNT**$(\mathcal{R})$ for clarity. Given any partial route R' which starts at v_j and ends at d, R_1^* and R_2^* can be connected to form two complete routes R_1 and R_2. 1) For ∃**R*k*NNT**, if $|\omega^*(R_1^*)| > |\omega(R_2^*)|$, then $|\omega(R_1)| \geq |\omega^*(R_1^*)| + |\omega(R')|$, as there is no intersection between $\omega^*(R_1^*)$ and $\omega(R')$ because $\mathcal{T} \in \omega^*(R_1^*)$ is the set of transitions that have ***k*NN** in R_1^* for both origin and destination points. Given that $|\omega(R_2)| \leq |\omega(R_2^*)| + |\omega(R')|$, $|\omega(R_1)| > |\omega(R_2)|$, while $\psi(R_1^*) < \psi(R_2^*)$, 2) ∀**R*k*NNT**, $|\omega^*(R_2)| \leq |\omega(R_2^*)| + |\omega^*(R')|$, while

$|\omega^*(R_1)| \geq |\omega^*(R_2^*)| + |\omega^*(R^{'})|$, so $|\omega^*(R_1)| > |\omega^*(R_2)|$. Without further spreading, we can see the priority relationship between $|\omega(R_1^*)|$ and $|\omega(R_2^*)|$ holds. □

In Algorithm 6, a dynamic table $\mathcal{DT}$ is maintained to store the pairs for every vertex accessed, and updates continue when new feasible partial routes are explored during the search. This is used to compare the **R*k*NNT** set and the travel distance of partial routes. The entry for a vertex v inserts a partial route R^* which ends at v when an existing partial route cannot be found which dominates R^*. After insertion, old entries in $\mathcal{DT}$ dominated by the new route R^* are removed. If a new one dominates R^*, the loop terminates and the next partial route is processed.

Example 6. *In Figure 7, $\{\{a\}, 0, 20\}$ is added to the queue $\mathcal{Q}$ after checking the reachability from a to j by comparing the pre-computed shortest distance with τ. Then, pop the queue $\mathcal{Q}$ to get the partial route R. Next, the last point a of R is checked to see if its neighbor b can be reached, and it can since $\psi(\overline{bej}) = 3.8 < (6 - 1.6)$. So $\{\{a, b\}, 1.6, \{T_1\}\}$ is added to $\mathcal{Q}$. Similarly, $\{a, c\}$ is inserted into $\mathcal{G}$. $\{a, d\}$ cannot be enqueued as the shortest distance from d to j is $\psi(\overline{dfhj}) = 5.2 > (6 - 1)$. $\{\{a, b, e\}, 3.1, \{T_1, T_2\}\}$ and $\{\{a, c, e\}, 2.6, \{T_1, T_2, T_3, T_4\}\}$ are enqueued and $\mathcal{DT}[e] = \{\{a, b, e\}, 3.1, \{T_1, T_2\}\}$ is updated. Further, $\{\{a, c, f, h\}, 3.9, \{T_1, T_2, T_3, T_4\}\}$ is enqueued. $\{\{a, b, e, h\}, 4.5, \{T_1, T_2\}\}$ has a greater travel distance, and $\omega(\overline{abeh}) = \{T_1, T_2\}$, and $\omega^*(\overline{acfh}) = \{T_1, T_2, T_3, T_4\}$, so $|\omega^*(\overline{acfh})| > |\omega(\overline{abeh})|$, and $\overline{acfh}$ dominates $\overline{acfh}$. Based on this extension in the graph, when $\mathcal{Q}$ is empty, the algorithm terminates.*

For **MinR*k*NNT**, Line 8 is changed to $max \leftarrow \infty$, and Line 23 is changed to $|\omega(R^*)| < max$. Moreover, one additional check called checkBounds$(max, \omega(R^*))$ after Line 14 in Algorithm 6 must be added. Given a partial route R^* and the existing optimal route R and max, R^* can be discarded when $|\omega(R^*)| > max$ as R^* cannot beat the existing optimal route R.

7 Experiments

7.1 Setup

We performed experiments to evaluate our solutions for **R*k*NNT** and **MaxR*k*NNT** using real bus route data and check-in data from Foursquare[3] in New York and Los Angeles, which are two largest cities in the USA. Moreover, we further produced a synthetic dataset which is normally distributed in NYC. We have published our dataset[4] to improve the reproducibility of our results. Figure 8 shows the heatmap of the route and check-in datasets. All experiments were performed on a machine using an Intel Xeon E5 CPU with 256 GB RAM running RHEL v6.3 Linux, implemented in C++, and compiled using GCC 4.8.1 with -O2 optimization enabled. The node size of **TR-tree** and **RR-tree** is set as 4KB, which means that a single node may contain hundreds of child nodes at most.

TABLE 2
Route Datasets.

Dataset	$\lvert\mathcal{D}_\mathcal{R}\rvert$	$\lvert\mathcal{G}.E\rvert$	$\lvert\mathcal{G}.V\rvert$
LA-Route	1,208	72,346	14,119
NYC-Route	2,022	61,118	16,999

3. https://foursquare.com/
4. https://sites.google.com/site/shengwangcs/home/rknnt

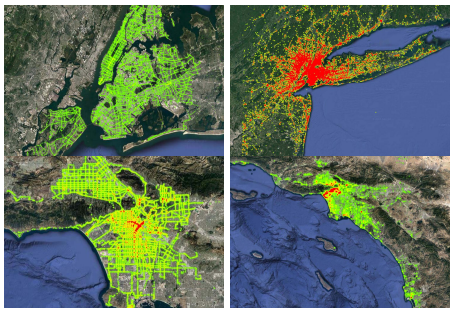


Fig. 8. The heatmap of the bus route dataset (left) and the transition dataset (right) in NYC (up) and LA (down).

TABLE 3
Transition Datasets.

Dataset	$\lvert\mathcal{D}_\mathcal{T}\rvert$	Latitude	Longitude
LA-Transit	109,036	[32°,35°]	[−120°,−117°]
NYC-Transit	195,833	[39°,42°]	[−75°, −72°]
NYC-Synthetic	10,000,000	[39°,42°]	[−75°, −72°]

TABLE 4
Parameter Settings.

Parameter	Value
$\lvert Q\rvert$	3,4,5,6,7,8,9,10
k	1,5,10,15,20,25
$\mathcal{I}$: Interval of route	$1km$, $2km$, $3km$, $4km$, $5km$, $6km$
$\psi(se)$: Length of route	$10km$, $20km$, $30km$, $40km$, $50km$
$\frac{\tau}{\psi(se)}$: Ratio of travel distance	1, 1.2, 1.4, 1.6, 1.8, 2.0

Route Datasets. We use two real bus network datasets, namely *NYC-Route* and *LA-Route*. We extracted the data from the GTFS datasets of New York[5] and Los Angeles[6]. Table 2 provides a breakdown of each dataset.

Transition Datasets. Two real transition datasets, *NYC-Transit* and *LA-Transit*, were produced by cleaning the Foursquare check-in data [3], and statistics for the cleaned data is shown in Table 3. Specifically, we divided a user's trajectory with multiple points into several transitions with two points. A trajectory with n points can be divided into $n-1$ transitions. Since the real dataset is small, we also generated a synthetic dataset which contains 10 million transitions for the NYC dataset, and refer to it as *NYC-Synthetic*.

7.2 Evaluation of R*k*NNT

Algorithms for evaluation. We compared the following methods when processing **R*k*NNT** over the two datasets. (1) **Filter-Refine (FO)**: the basic framework proposed in Section 4; (2) **Voronoi (VO)**: the Voronoi-based method which can create a larger filtering area by drawing a Voronoi diagram based on the query and filtering route after regular filtering by points; (3) the **Divide-Conquer (DC)** method which is proposed in Section 5.2.

Queries. We prepared two query sets: the first set is a synthetic query set for the purposes of parameter evaluation, and generated as follows: 1) We randomly generated $1,000$ points from $\mathcal{D}_\mathcal{R}$. 2) We iteratively chose each point as a start point, and append new points one by one with a limited rotation angle to simulate a realistic case. The rotation angle of every time extension does not exceed $90°$, so the query route will not zigzag [9]. All experimental results are reported as the mean of all $1,000$ queries.

5. http://web.mta.info/developers/developer-data-terms.html#data
6. http://developer.metro.net/gtfs/google_transit.zip

The second query set contains all the routes in *NYC-Transit* and *LA-Transit*, which are used as queries to test our most efficient method, **Divide-Conquer**.

Parameters. Table 4 summarizes all key parameters for a query, and the default values are underlined. $\mathcal{I} = \frac{\psi(Q)}{|Q|}$ is the interval length between two adjacent points in the query, where $\psi(Q)$ is the travel distance of the query route computed by Equation 7.

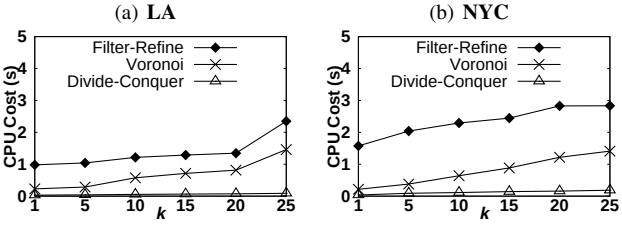


Fig. 9. Effect on Running Time with the increasing of k.

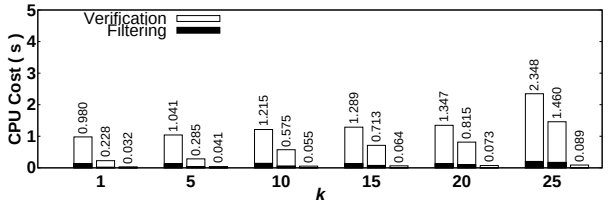


Fig. 10. Breakdown of running time with increasing k in **LA**.

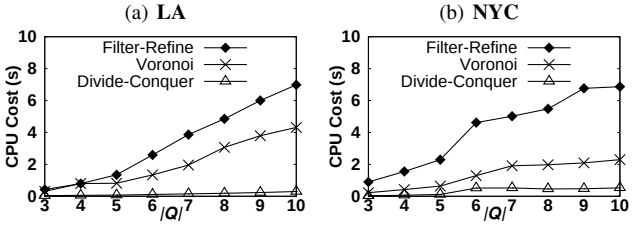


Fig. 11. Effect on running time with the increasing of $|Q|$.

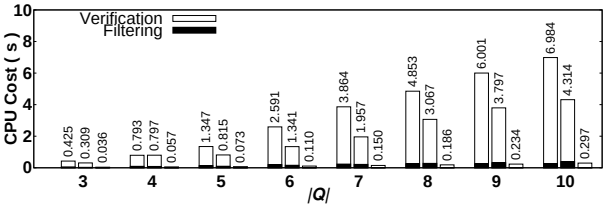


Fig. 12. Breakdown of running time w.r.t. $|Q|$ in *LA*.

Effect of $|Q|$. Figure 11 shows the running time of our three methods. As more points are added into the query, **Filter-Refine** and **Voronoi** exhibit a sharp increase in running time. Since these methods need more time to check whether a node can be filtered, the filtering space becomes smaller and the probability of being pruned decreases. In contrast, **Divide-Conquer** shows almost a linear increase. This is probably a result of the whole query being divided into $|Q|$ queries, and a node is not be pruned by checking every query point. Figure 12 shows a breakdown of the running time to the tasks of filtering and verification on the LA data, and the verification occupies more than 80% for most cases.

Effect of k. Figure 9 shows that the time cost for all three methods will increases as k increases. This is because it is unlikely that a point can be filtered by k filtering routes when k is large.

Effect of $\mathcal{I}$. We observe that the intervals $\mathcal{I}$ between two adjacent points vary from route to route in real life. Hence, we conducted experiments to see how the running time is affected in this scenario. As mentioned when describing query generation, the size of the query is increased by appending randomly generated points, one at a time. Figure 13(a) and Figure 13(b) show that there is a slight increase on the running time when $\mathcal{I}$ is large. The main reason is that when two query points are close, a node can be filtered by a filtering point easily, while when the intervals are large, it is harder to filter a node.

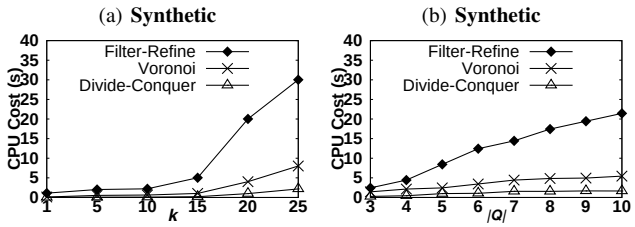


Fig. 13. Effect on running time with the increasing of $|Q|$ and k in synthetic dataset.

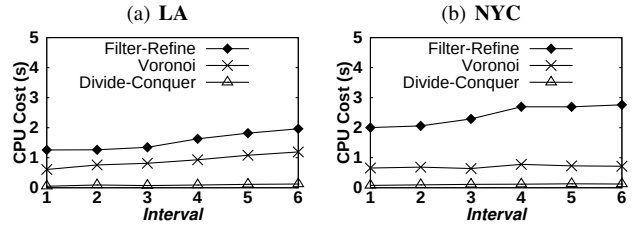


Fig. 14. Effect on running time with the increasing of $\mathcal{I}$.

Real Route Queries. After testing the effect of each individual parameter, we took every route in each dataset as a query to evaluate our best method **Divide-Conquer**. Note that before running each query, we removed the points of this route from the **RR-tree** index. Figure 16 shows that over 90% of the queries can be processed in less than 5s. The main reason is the relationship to the number of points in the query.

Index Update. The time cost of index creation and updates is shown in Table 5. We inserted 10 to 50 thousand transition points into the index, respectively, and observe that every insertion costs less than 1 ms even when using the largest dataset – *Synthetic*. The main reason is that the route dataset is fixed, so **RR-tree**, **NList** and **PList** are also fixed, and only **TR-tree** must be updated when transitions are inserted. Therefore, the main cost when performing updates comes from **TR-tree**.

TABLE 5
Index Update Performance.

Dataset	$\lvert\mathcal{D}_\mathcal{T}\rvert$	+10k	+20k	+30k	+40k	+50k
LA	2.21s	0.56s	0.76s	1.88s	2.32s	2.62s
NYC	3.10s	0.65s	1.44s	1.65s	2.33s	2.83s
Synthetic	13m18.56s	4.37s	5.82s	6.35s	7.33s	8.65s

I/O & Pruning. Table 6 shows the the number of page accesses in **RR-tree** and **TR-tree** for all three methods over two datasets with various parameters. The filtering effectiveness is shown on the right side of table. We can see that the I/O increases as k and $\mathcal{I}$ increase, and $|Q|$ affects it the most (see the bold numbers in 3^{rd} column). Interestingly, we also find that more filtering nodes and points do not always translate to higher efficiency (see the bold numbers), as an efficient algorithm such as **DC** always prunes upper-level nodes early, while non-efficient algorithms cannot achieve early stage pruning, but access more lower-level child nodes or points. In summary, our main observations are:

1) **Divide-Conquer** consistently has the best performance in terms of running time and I/O, followed by **Voronoi**, with **Filter-Refine** being the worst.

TABLE 6
Statistics of I/O and Pruning Effectiveness.

Parameter		I/O (#accessed nodes)						Pruning nodes/points in TR-tree					
		LA			*NYC*			*LA*			*NYC*		
		FR	**VO**	**DC**	**FR**	**VO**	**DC**	**FR**	**VO**	**DC**	**FR**	**VO**	**DC**
k	1	66	58	58	85	85	78	**938/2423**	**922/3351**	**2554/4543**	841/3131	977/4224	2695/6779
	5	90	71	71	111	90	90	950/2477	918/3518	**2642/4896**	850/3209	1002/4438	2852/7276
	10	614	418	339	842	606	466	934/2496	916/3576	2750/5052	839/3259	1009/4600	2950/7728
$\|Q\|$	3	**406**	221	206	71	61	61	928/1999	1002/3238	1676/2786	651/1983	713/1834	829/1743
	7	**818**	666	466	222	122	122	2104/3764	2234/5274	3754/7374	723/4293	1239/5213	2157/8129
	10	**1189**	1021	594	268	128	128	4016/8912	4761/10692	5281/12049	751/4542	1451/7245	2945/9046
$\mathcal{I}$	1	543	351	305	108	88	88	918/2436	921/3435	2702/4638	841/3251	1021/4561	2903/7521
	3	602	398	330	111	90	90	936/2481	911/3556	2701/4983	852/3277	990/4552	2726/7550
	6	697	563	406	119	94	94	947/2790	913/3642	2716/5752	854/3315	962/4433	2493/7578

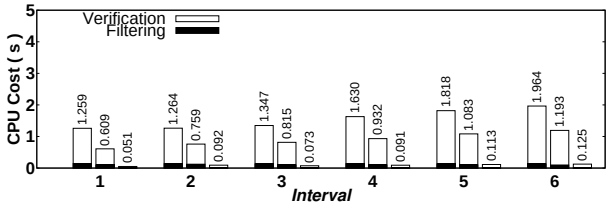


Fig. 15. Breakdown of running time with increasing $\mathcal{I}$ in *LA*.

(a) **LA** (b) **NYC**

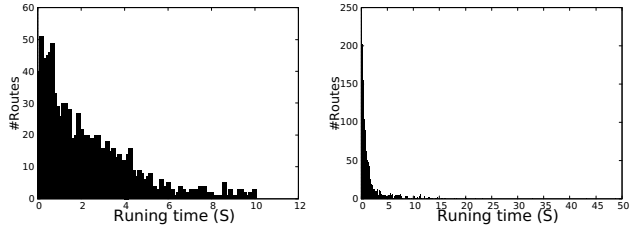


Fig. 16. The distribution of running time when taking all existing bus routes as query by **MaxRkNNT** when $k = 10$.

2) All three methods are sensitive to k and $|Q|$. Only **Filter-Refine** and **Voronoi** are sensitive to the interval length $\mathcal{I}$ of the query.
3) When taking existing routes as real queries, most queries can be answered efficiently using **Divide-Conquer**.

7.3 Evaluation of MaxRkNNT

Algorithms for evaluation. (1) **BruteForce**: the baseline method which uses the *k shortest paths* [32] to find all the routes which have a smaller travel distance than the distance threshold τ, after which an **RkNNT** is performed on every candidate to choose the maximal one. (2) **Pre**: the method that extends the **BruteForce** by pre-computation of the **RkNNT** set for every vertex without an on-the-fly **RkNNT** query. For **MaxRkNNT** and **MinRkNNT**, both can be solved using the same pruning techniques with little difference in bound checking, which has a small impact on performance. We denote them as (3) **Pre-Max** and (4) **Pre-Min**.

Queries. To test the effect of key parameters, we first generated a point set by choosing $1,000$ start points randomly from our route datasets. Then, we searched 6 end points for every start point with different $\psi(se)$, which is the distance between the origin and the destination, as shown in the last row of Table 4. Furthermore, we used existing representative routes as queries and employed **MaxRkNNT** and **MinRkNNT** search algorithm to find the new "optimal" routes. Finally, we compared the **RkNNT** sets of the original routes against the new routes.

Parameters. We discovered two key parameters that affect the performance of **MaxRkNNT**: (1) the coverage degree of a bus route - denoted by $\psi(se)$ and quantified as the Euclidean distance between the start and end points of a query Q. (2) $\frac{\tau}{\psi(se)}$, which is the ratio of the travel distance over the straight-line distance from origin to destination of Q. The choices of these parameters are from the distribution of all real bus routes, as shown in Figure 17.

Pre-computation. Table 7 shows the time spent on pre-computation, which is composed of **RkNNT** search and all-pair shortest distance computations, the bold numbers are the results for synthetic dataset. The pre-computation consists of of two steps: the **RkNNT** query for every vertex, and the shortest distance route search, as shown in Algorithm 5. All-pair shortest distance computation costs about 4 minutes for both datasets, and the **RkNNT** search of all vertices in $\mathcal{G}$ costs less than 5 minutes when $k = 10$. For the synthetic dataset, the time spent on pre-computation is about 12 minutes when $k = 10$.

Effect of $\psi(se)$. Figure 17(a) shows that the time spent on the search task increases when the distance between the origin and destination $\psi(se)$ increases. This is because more vertices in the graph need to be scanned between the origin and destination. For **Bruteforce**, the reasons are twofold: (1) It returns more candidate routes for **RkNNT**; (2) The candidate routes are longer when $\psi(se)$ is long, so more time has to be spent for every **RkNNT** query. In contrast, for the remaining three methods, since we have pre-computed the **RkNNT** set for every vertex, the running time comes from the search over $\mathcal{G}$. **Pre-Max** has the best performance due to the bound checking during the spreading of partial routes.

Effect of $\frac{\tau}{\psi(se)}$. To generate the query, we choose a subset of queries with a fixed $\psi(se)$ as the default value shown in Table 4 and alter τ in the experiment. Figure 19 shows that increasing $\frac{\tau}{\psi(se)}$ leads to an increased running time, it can also be ascribed to the more candidates between the origin and destination.

Real queries. We took each route in $\mathcal{D}_{\mathcal{R}}$ as a query to perform a **MaxRkNNT** search to see whether we can find a better route which has a larger **RkNNT** set while maintaining an acceptable travel distance threshold. Each query is generated using the start and end bus stop, and the travel distance for each route. Figure 20 shows the running time distribution for the real queries, most

TABLE 7
Running time(s) for Pre-computation when $k = 1, 5, 10$.

k	*LA*			*NYC*		
	1	5	10	1	5	10
RkNNT	80.5	153.2	230.8	140.4	202.1	253.5
				201.7	**545.8**	**748.1**
Shortest		191.3			251.9	

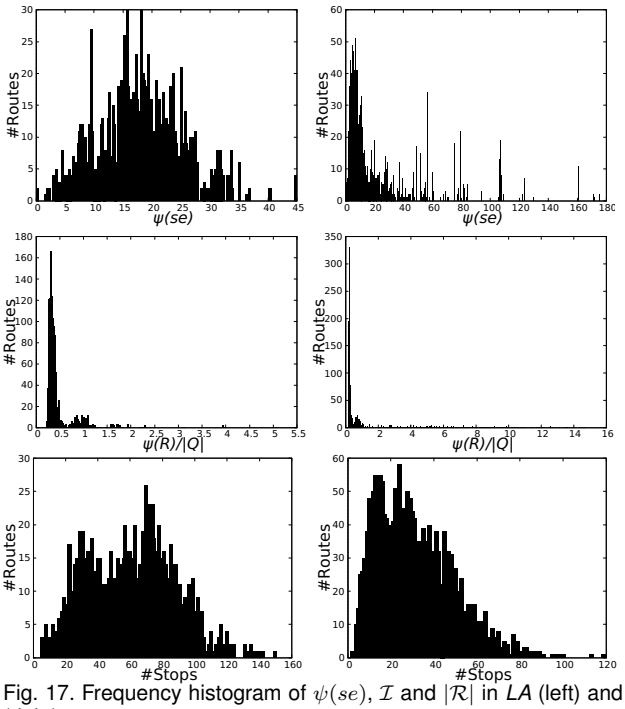


Fig. 17. Frequency histogram of $\psi(se)$, $\mathcal{I}$ and $|\mathcal{R}|$ in *LA* (left) and *NYC* (right).

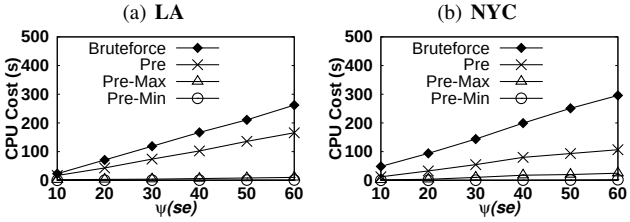


Fig. 18. Effect on running time as $\psi(se)$ increases.

queries in the LA data can be answered in less than a second.

In Figure 21, we show four kinds of routes which share the same start and end locations: 1) the original bus route passes through Manhattan, 2) the shortest distance route, 3) the **MaxR*k*NNT** route which attracts the most passengers, 4) the **MinR*k*NNT** route which attracts the fewest passengers. The right table shows the search time, number of passengers, travel distance, and number of stops. We find: (1) the original route and the **MaxR*k*NNT** route are almost the same (in particular, **MaxR*k*NNT** finds a route which just is 10 meters longer but can attract 129 extra passengers), which means that the existing bus route is almost optimal between the start and end stops. (2) If a driver wants to save time, the least crowded route can be selected as provided by **MinR*k*NNT**; if the car should be shared to increase revenue, the route found by **MaxR*k*NNT** is better.

8 Conclusion

In this paper, we proposed and studied the **R*k*NNT** query, which can be used directly to support capacity estimation in bus networks. First, we proposed a filter-refine processing framework, and an optimization to increase the filtering space that improves pruning efficiency. Then we employed **R*k*NNT** to solve the bus route planning problem. In a bus network, given a start and end bus stop, we can find an optimal route which attracts the most passengers for a given travel distance threshold. To the best of our knowledge, this is the first work studying reverse k nearest neighbors in trajectories, and our solution supports dynamically changing transition data while providing up-to-date answers efficiently. **R*k*NNT** can be used in other related applications such as route planning for emergency vehicles, traffic evacuation.

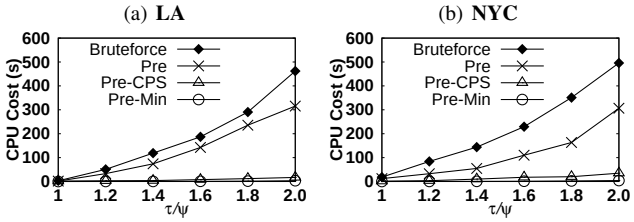


Fig. 19. Effect on running time with the increase of $\frac{\tau}{\psi(se)}$.

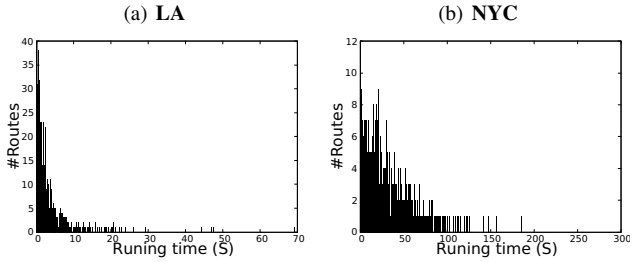


Fig. 20. Distribution of running time of **MaxR*k*NNT** on real route query.

	ST (s)	**NP**
1	N/A	1,032
2	0.004	817
3	1.02	1,161
4	0.31	713
	TD (m)	**#Stops**
1	10,238	49
2	9,012	43
3	10,248	49
4	9,543	40

Fig. 21. Comparison among four routes: ST (searching time), NP (number of passengers), TD (travel distance) and the number of stops.

Acknowledgments

This work was partially supported by ARC DP170102726, DP170102231, DP180102050, and National Natural Science Foundation of China (NSFC) 61728204, 91646204. Zhifeng Bao is supported by a Google Faculty Award.

References

[1] A. Abadi, T. Rajabioun, and P. A. Ioannou. Traffic flow prediction for doad transportation networks with limited traffic data. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):653–662, 2015.

[2] H. Aljazzar and S. Leue. K*: A heuristic search algorithm for finding the k shortest paths. *Artificial Intelligence*, 175(18):2129–2154, 2011.

[3] J. Bao, Y. Zheng, and M. F. Mokbel. Location-based and preference-aware recommendation using sparse geo-social networking data. In *SIGSPATIAL*, pages 199–208, 2012.

[4] M. A. Bolívar, L. Lozano, and A. L. Medaglia. Acceleration strategies for the weight constrained shortest path problem with replenishment. *Optimization Letters*, 8(8):2155–2172, 2014.

[5] G. Casanova, E. Englmeier, M. E. Houle, M. Nett, E. Schubert, and A. Zimek. Dimensional testing for reverse k-nearest neighbor search. *PVLDB*, 10(7):769–780, 2017.

[6] A. Ceder. *Public Transit Planning and Operation*. 2007.

[7] M. A. Cheema, X. Lin, W. Zhang, and Y. Zhang. Influence zone: Efficiently processing reverse k nearest neighbors queries. In *ICDE*, pages 577–588, 2011.

[8] M. A. Cheema, W. Zhang, X. Lin, Y. Zhang, and X. Li. Continuous reverse k nearest neighbors queries in Euclidean space and in spatial networks. *VLDB Journal*, 21(1):69–95, 2012.

[9] C. Chen, D. Zhang, N. Li, and Z. H. Zhou. B-planner: Planning bidirectional night bus routes using large-scale taxi GPS traces. *IEEE Transactions on Intelligent Transportation Systems*, 15(4):1451–1465, 2014.

[10] T. Emrich, H. P. Kriegel, N. Mamoulis, J. Niedermayer, M. Renz, and A. Züfle. Reverse-nearest neighbor queries on uncertain moving object trajectories. In *DASFAA*, pages 92–107, 2014.

[11] V. Guihaire and J.-K. Hao. Transit network design and scheduling: A global review. *Transportation Research Part A: Policy and Practice*, 42(10):1251–1273, 2008.

[12] A. Guttman. R-trees: a dynamic index structure for spatial searching. *ACM SIGMOD Record*, 14(2):47–57, 1984.

[13] Y. Liu, C. Liu, N. J. Yuan, L. Duan, Y. Fu, H. Xiong, S. Xu, and J. Wu. Intelligent bus routing with heterogeneous human mobility patterns. *Knowledge and Information Systems*, 50(2):383–415, 2016.

[14] L. Lozano and A. L. Medaglia. On an exact method for the constrained shortest path problem. *Computers and Operations Research*, 40(1):378–384, 2013.

[15] Y. Lv, Y. Duan, W. Kang, Z. Li, and F. Y. Wang. Traffic flow prediction with big data: a deep learning approach. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):865–873, 2014.

[16] E. Q. V. Martins and M. M. B. Pascoal. A new implementation of Yen's ranking loopless paths algorithm. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1(2):121–133, 2003.

[17] F. Pinelli, R. Nair, F. Calabrese, M. Berlingerio, G. Di Lorenzo, and M. L. Sbodio. Data-driven transit network design from mobile phone trajectories. *IEEE Transactions on Intelligent Transportation Systems*, 17(6):1724–1733, 2016.

[18] R. Seidel. On the all-pairs-shortest-path problem. In *STOC*, pages 745–749, 1992.

[19] S. Shang, L. Chen, Z. Wei, C. S. Jensen, K. Zheng, and P. Kalnis. Trajectory similarity join in spatial networks. *PVLDB*, 10(11):1178–1189, 2017.

[20] S. Shang, R. Ding, B. Yuan, K. Xie, K. Zheng, and P. Kalnis. User oriented trajectory search for trip recommendation. In *EDBT*, pages 156–167, 2012.

[21] S. Shang, B. Yuan, K. Deng, K. Xie, and X. Zhou. Finding the most accessible locations: reverse path nearest neighbor query in road networks. In *SIGSPATIAL*, pages 181–190, 2011.

[22] A. Shivraj and S. Ia. Urban bus transit route network design using genetic algorithm. *Journal of Transportation Engineering*, 124(4):368–375, 1997.

[23] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.

[24] Y. Tao, D. Papadias, and X. Lian. Reverse kNN search in arbitrary dimensionality. In *PVLDB*, pages 744–755, 2004.

[25] J. L. Toole, S. Colak, B. Sturt, L. P. Alexander, A. Evsukoff, and M. C. González. The path most traveled: Travel demand estimation using big data resources. *Transportation Research Part C: Emerging Technologies*, 58, Part B:162–177, 2015.

[26] N. van Oort, T. Brands, and E. de Romph. Short-term prediction of ridership on public transport with smart card data. *Transportation Research Record: Journal of the Transportation Research Board*, 2535:105–111, 2015.

[27] S. Wang, Z. Bao, J. S. Culpepper, T. Sellis, M. Sanderson, and X. Qin. Answering top-k exemplar trajectory queries. In *ICDE*, pages 597–608, 2017.

[28] S. Wang, X. Xiao, Y. Yang, and W. Lin. Effective indexing for approximate constrained shortest path queries on large road networks. *PVLDB*, 10(2):61–72, 2016.

[29] W. Wu, F. Yang, C.-y. Chan, and K.-L. Tan. FINCH : evaluating reverse k-nearest-neighbor queries on location data. *PVLDB*, 1(1):1056–1067, 2008.

[30] O. Y. Xian, M. Chitre, and D. Rus. An approximate bus route planning algorithm. In *SSCI*, pages 16–24, 2013.

[31] S. Yang, M. A. Cheema, X. Lin, and W. Wang. Reverse k nearest neighbors query processing: experiments and analysis. *PVLDB*, 8(5):605–616, 2015.

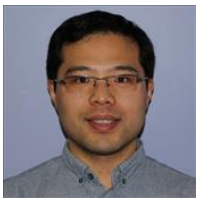
[32] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

[33] J. Zhang, D. Shen, L. Tu, F. Zhang, C. Xu, Y. Wang, C. Tian, X. Li, B. Huang, and Z. Li. A real-time passenger flow estimation and prediction method for urban bus transit systems. *IEEE Transactions on Intelligent Transportation Systems*, 18(11):3168–3178, 2017.

[34] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from GPS trajectories. In *WWW*, pages 791–800, 2009.

[35] Z. Zhou, W. Wu, X. Li, M. L. Lee, and W. Hsu. MaxFirst for MaxBRkNN. In *ICDE*, pages 828–839, 2011.

Sheng Wang received the BE degree in information security, ME degree in computer technology from Nanjing University of Aeronautics and Astronautics, China in 2013, 2016, respectively. He is currently working toward the PhD degree in the School of Science, RMIT University, Australia. His research interests mainly include spatial database, spatial-textual search, and trajectory management.



Zhifeng Bao received the PhD degree in computer science from the National University of Singapore in 2011. He is an assistant professor at RMIT University in Australia. His research interests include data usability, spatial database, data integration and cleaning.

J. Shane Culpepper completed a PhD at The University of Melbourne in 2008. He is now a Vice-Chancellor's Senior Research Fellow at RMIT University, where he runs the Information Discovery Research Lab. His research interests include information retrieval, text indexing, system evaluation, learning to rank, algorithm engineering, and scalable distributed/parallel computing.

Timos Sellis received the PhD degree in computer science from the University of California, Berkeley, in 1986. He is a professor and director of the Data Science Research Institute at Swinburne University of Technology, Australia. Between 2013 and 2015, he was a professor at RMIT University, Australia, and before 2013 the director of the Institute for the Management of Information Systems (IMIS) and a professor at the National Technical University of Athens, Greece. His research interests include big data, data streams, personalization, data integration, and spatio-temporal database systems. He is a fellow of the IEEE and ACM.

Gao Cong received the PhD degree from the National University of Singapore, in 2004. He is an associate professor with Nanyang Technological University, Singapore. Before he relocated to Singapore, he worked with Aalborg University, Microsoft Research Asia, and the University of Edinburgh. His current research interests include geo-textual data management and data mining.