Reverse $k$ Nearest Neighbor Search over Trajectories

Sheng Wang†, Zhifeng Bao‡, J. Shane Culpepper†, Timos Sellis†, Gao Cong‡
†RMIT University, ‡Swinburne University of Technology, †Nanyang Technological University
firstname.surname@rmit.edu.au †tsellis@swin.edu.au ‡gaocong@ntu.edu.sg

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 $D_R$, a set of passenger transitions $D_T$, 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 $\text{Max}RkNNT$ ($\text{Min}RkNNT$), 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 pre-defined travel distance threshold. Experiments on real datasets demonstrate the efficiency and scalability of our approaches. To the best of our best knowledge, this is the first work to study the $RkNNT$ problem for route planning.

1. INTRODUCTION

Reverse $k$ Nearest Neighbor ($RkNN$) queries have attracted considerable attention [13][16][18][6][21][20]. An $RkNN$ query aims to find all the points among a set of points that take a query point as their $k$ nearest neighbors. The $RkNN$ query has many applications in resource allocation, decision support and profile-based marketing, etc. For example, $RkNN$ can be applied to estimate the number of customers of a planned restaurant among all existing restaurants.

The increasing prevalence of GPS-enabled devices provides new opportunities to obtain many real trajectory data in a short period of time, such as the GPS trajectories of taxi or uber drivers, check-in trajectories of social media users in Foursquare, which describe the movements/transitions of people. In this paper, we will explore the $RkNN$ search over multiple-point trajectories (which is referred to as $RkNNT$). In a nutshell, the $RkNNT$ 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 $D_R$ to travel. Here, a passenger’s movement is modeled as a combination of an origin and a destination [10] such as home and office, which is called a transition. As shown in Figure 1, two red points represent a single transition $T$ of a passenger. Among the two available routes, $T$ would favour Route 2 as both points in $T$ can find a point closer in Route 2 than in Route 1.

The main difference between our problem and $RkNNT$ is that our query is a route, and our data collections contain routes and user transitions respectively. $RkNNT$ can estimate the passengers that will take the query route to travel. Another significant difference of our $RkNNT$ from previous work is that the transition data is very dynamic and new transitions will arrive continuously, such as the request of uber passengers. Therefore, it is important to take this into consideration in the solution to answering the $RkNNT$ query.

![Figure 1: Bus Routes and User Transition.](image-url)

$RkNNT$ serves as a core yet fundamental operator in many applications in transportation field. The first and foremost one is to estimate the capacity of a route based on passengers’ dynamic movements. On top of capacity estimation, $RkNNT$ can be used for the following promising applications.

1) Optimal Route Planning. (i) Among a set of candidate routes, an $RkNNT$ can be used to find the optimal route which has the maximum (minimum) number of passengers among a set of candidate routes. For Uber drivers, finding a route with the maximum number of passengers can increase profitability (the uber 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 can increase response times in emergency situations. We will address problem (i) in this paper as well. (ii) Furthermore, by taking the temporal factor into consideration, i.e., user transitions at different time periods, it can help further adjust the frequency of planned vehicles on the planned routes, in order to save running cost for either individual vehicle drivers or public transportation authority [5].

The first challenge in answering $RkNNT$ lies in how to prune the transitions which cannot be in the results without accessing every transition. A straightforward method is to conduct a $k$NN search for every transition, and then check the resulting ranked
lists to see whether the query is a $k$NN. This method is intractable when there are a large number of transitions and new transitions are being added to the database. For MaxR$k$NN, 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$k$NN 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$k$NN, it is crucial to prune out candidates which cannot be an optimal route. Another challenge is how to support dynamic updates as old transitions expire and new transitions arrive.

To address this challenge, we first build two R-tree indexes RR-tree and TR-tree 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 route index 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 can not have the query as a nearest neighbor. After finding the filtering set, the pruning of transitions starts by traversing TR-tree and checking if a node can be filtered by more than $k$ routes in the filtering set. Finally, all of the candidate transitions are verified using the filtered nodes during the traversal of RR-tree.

Next, we explore the optimal route planning problem, where we consider a graph formed by the bus network. Given a starting location and a destination, we find the optimal route $R$ which connects the two locations in a bus network, and maximizes (minimize) the number of passengers that take $R$ as its $k$NN without exceeding a distance threshold. We call this problem as the MaxR$k$NN (MinR$k$NN).

2) Bus Advertisement Recommendation. As an R$k$NN query for a route can locate the set of passenger transitions who would take the travel route, and we can obtain the profiles of potential passengers using social networks, a deeper analysis of the common interests among passengers who take similar travel routes can be found. Consequently, we can select and broadcast advertisements that will have the largest influence on passengers taking a route.

For MaxR$k$NN, a weighted graph is built by the pre-computed R$k$NN set for every vertex. Then 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 distance 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$k$NN problem for the first time, which serve as a core yet frequently adopted operator in many practical applications. In particular, we explore how to use R$k$NN to plan an optimal route which attracts the most (or fewest) passengers (Section 5).
- 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 also introduce MaxR$k$NN (MinR$k$NN) 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).

2. PRELIMINARY & 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 employ a pruning-refinement frameworks to avoid scanning the entire dataset. However, these approaches cannot easily be translated to route search where both queries and collections are of multi-point trajectories. Given a set of candidates as a query, a maximizing R$k$NN query finds the optimal trajectory which satisfies the maximum number of results $[17] [24]$.

How to improve the search performance has attracted much attentions over last decades. The basic intuition behind filtering out a point $p$ is to find another point which is closer to $p$ than the query point $q$. 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 would 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 query, Shang et al. $[14]$ 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. $[17]$ proposed a continuous reverse nearest neighbors query to monitor a moving object and find all static points that take the moving object query as a $k$ nearest neighbor. Recently, Emrich et al. $[9]$ solved a 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 for every time stamp 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 Searching

Bus Route Planning. Bus network design is known to be a complex, non-linear, non-convex, multi-objective NP-hard problem. Based on existing bus networks, Pattnaik et al. $[13]$ proposed a heuristic method which uses a genetic algorithm to minimize the cost of passengers and operators. Yang et al. $[22]$ used ant colony algorithms to maximize the number of direct travelers between two nearby bus stops. Population estimation and user surveys $[19]$ around the planned route are traditional ways to estimate the number of passengers that may use the planned travel route. 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.
Chen et al. [8] 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 a bus route graph based on the connectivity between two stops. Based on human mobility patterns, Liu et al. [10] proposed a localized transportation choice model, which can predict bus travel demand for different bus routes by taking into account both bus and taxi travel demands. However, the method must scan the static records for all of the customers, which is inefficient in practice, and the model has to be rebuilt whenever the records are updated.

**Shortest Route Searching.** Given a starting vertex and an ending vertex, the classical route planning 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 ($k$SP) [23][1], 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 [23] 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 [3] applies resource constraints on each edge, and solves the shortest path search problem based on these constraints. An example constraint would be time costs.

### 3. PROBLEM DEFINITION

In this section, we formally define the $R\&NNT$ problem and important notations are recorded in Table 1

**Definition 1. (Route)** A route $R$ 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 $T$ contains an origin point $o$ and a destination point $d$.

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

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

$$\text{dist}(t, R) = \min_{r \in R} \text{distance}(t, r)$$

(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 $\mathcal{D}_R$, the $k$NN search of a transition point $t \in \mathcal{T}$ retrieves a set $S \in \mathcal{D}_R$ of $k$ routes such that for all $R \in S$, and for all $R' \in \mathcal{D}_R - S$: $\text{dist}(t, R) \geq \text{dist}(t, R')$.

In particular, two types of $k$NN are supported for a transition $T$, which can also be found in [9].

1. $\exists k$NN: $T$ takes $R$ as a $k$NN iff there exists a point $t \in T$ taking $R$ as $k$NN. So, $\exists k$NN$(T) = k$NN$(t_o) \cup k$NN$(t_d)$.

2. $\forall k$NN: $T$ takes $R$ as a $k$NN iff both points $t_o$ and $t_d$ take $R$ as their $k$NN. So, $\forall k$NN$(T) = k$NN$(t_o) \cap k$NN$(t_d)$.

### 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 $\{q_1, ..., q_m\}$ |
| $\mathcal{D}_R, \mathcal{D}_T$ | The route and transition sets |
| $\text{dist}(t, R)$ | Distance from transition point $t$ to $R$ |
| $\perp(q, r)$ | Perpendicular bisector between $q$ and $r$ |
| $H_{q,r}, H_{r,q}$ | Two half-planes divided by perpendicular bisector $\perp(q, r)$ |
| $H_{r,Q}, H_{R,Q}$ | Filtering space formed by $Q$ with $r$ and $R$ |
| $\mathcal{C}(r)$ | Crossover route set of $r$ |
| $S_{\text{filter}}, S_{\text{refine}}$ | Filtering set and filtered node set |
| $S_{\text{end}}, S_{\text{result}}$ | Transition candidates and result set |
| $\text{root}_x, \text{root}_y$ | Root of Route R-tree (Transition R-tree) |
| $\mathcal{V}_{R,Q}$ | Voronoi diagram formed by $R$ and $Q$ |
| $\mathcal{N}$ | Weighted graph |
| $\tau$ | Travel distance threshold |
| $\omega(\mathcal{R}), \psi(\mathcal{R})$ | R\&NNT set and travel distance of $R$ in $\mathcal{Q}$ |
| $\mathcal{M}_o[i][j]$ | Lower bound matrix of $\psi(\mathcal{R})$ where $R$ starts from vertex $i$ to $j$ |

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

**Definition 5. ($R\&NNT$)** Given a set of routes $\mathcal{D}_R$, a set of transitions $\mathcal{D}_T$, and a query route $Q$, $\forall R\&NNT(Q)$ retrieves all transitions $T \in \mathcal{D}_T$, such that for all $T$: $Q \in \exists k$NN$(T) \land k$NN$(T)$.

**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 $T_1$ and $T_2$ denote the origin point and destination point for transition $T_3$. The query route $Q$ is composed of 5 query points (in red). If we take the $\forall R\&NNT$ query, as point $T_3^+$ and $T_3^-$ take $Q$ as the nearest route, $T_4$ will be the result of $\forall R\&NNT(Q)$.

**Lemma 1.** Given a query $Q$, $\forall R\&NNT(Q) \subseteq \exists k$NN$(Q)$.

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

Using Lemma 1 the set of transition points which take $Q$ as a $k$NN can be searched for first, and then $\forall R\&NNT(Q)$ can be found by adding the corresponding routes. For $\forall R\&NNT$, we need to remove

---

**Figure 3: Example of routes and transitions.**
transitions that have only one point in $\exists \mathbb{R} \cap \mathbb{NNT}(Q)$. Hence, a unified framework can be proposed that answers both $\exists \mathbb{R} \cap \mathbb{NNT}$ and $\forall \mathbb{R} \cap \mathbb{NNT}$. In the rest of this paper, we use $\mathbb{R} \cap \mathbb{NNT}$ to represent $\exists \mathbb{R} \cap \mathbb{NNT}$ by default for ease of composition.

4. CAPACITY ESTIMATION - A PROCESSING FRAMEWORK FOR $\mathbb{R} \cap \mathbb{NNT}$

In this section, we first provide a sketch of our framework to answer the $\mathbb{R} \cap \mathbb{NNT}$ 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

All impossible transition points are pruned using a PruneTransition algorithm, and the remaining candidates $S_{\text{end}}$ are further verified using a RefineCandidates algorithm to generate the final result set $S_{\text{result}}$. Note that before pruning, a subset of routes $S_{\text{filter}}$ 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.

Algorithm 1: $\mathbb{R} \cap \mathbb{NNT}(Q, \text{root}_r, \text{root}_t)$

Output: $S_{\text{result}}$: the result set
1. $(S_{\text{filter}}, S_{\text{refine}}) \leftarrow \text{FilterRoute}($root$_r, Q, k)$; // Sec 4.2.1
2. $S_{\text{end}} \leftarrow \text{PruneTransition}(\text{root}_t, Q, S_{\text{filter}}, k)$; // Sec 4.2.2
3. $S_{\text{result}} \leftarrow \text{RefineCandidates}(Q, S_{\text{end}}, S_{\text{refine}});$ // Sec 4.2.3
4. return $S_{\text{result}}$

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 point in this transition can be pruned safely, so pruning transitions helps to find the filtering routes to prune the points in the transition.

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

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

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 points using the filtering point $r$ from the routes.

Recall the example in Figure 3 where an $\mathbb{R} \cap \mathbb{NNT}$ 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 $\mathbb{R}$ in $\mathbb{D}_R$; then based on the straight line $\triangle q_i r$, 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:

\[
H_{r-Q} = \bigcap_{q_i \in Q} H_{r-q_i}
\]

Figure 4: Pruning by half-space for a multi-point query $Q$.

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:

\[
H_{r-Q} = \bigcap_{q_i \in Q} H_{r-q_i}
\]

the point $r$ which belongs to $\mathbb{R} \in \mathbb{D}_R$ is called as the filtering point.

As shown in Figure 4, we can see that there are five perpendicular bisectors. They form a polyline $\text{abdec}$ 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 box (MBB) such as $MBR_5$ covering points $T_5^e$ and $T_5^o$ is located entirely inside the filtering space, then $T_5^e$ and $T_5^o$ inside this MBB will not take the query as a nearest neighbor and can be filtered out.

Every point in the route set $\mathbb{D}_R$ can be a filtering point, but we cannot choose all points in the route set $\mathbb{D}_R$ to do pruning especially when the whole set is large and located in external memory. To pruning a transition point, if we access all route points every time, the process is costly. 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

- $\text{RR-tree}$ & $\text{TR-tree}$ are two tree indexes for point data fetched from route dataset $\mathbb{D}_R$ and transition dataset $\mathbb{D}_T$ respectively, and referred to as a Route R-tree ($\text{RR-tree}$) and a Transition R-tree ($\text{TR-tree}$). The tree indexes are created first, and every point in the leaf node of $\text{RR-tree}$ contains the IDs of the routes it belongs to. Every point in $\text{TR-tree}$ also contains the IDs of the transition it belongs to. Through the transition ID and route ID in the node of $\text{RR-tree}$ and $\text{TR-tree}$, 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).

- $\text{NList}$. As we need to get all the routes that have a point inside a given node for verification in Section 4.2.2. Hence, for each node in $\text{RR-tree}$, we will create a list for every node of $\text{RR-tree}$ by traversing the whole tree bottom-up to store all the IDs of routes inside.
Our index supports dynamic updating, where new transitions and routes can be added into the index easily. This is in contrast the previous work [8][10] which needs to train whole dataset from scratch once there are new data inserted.

4.2 Key Functions

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

4.2.1 Filtering Routes

In order to get a smaller filtering set \( S_{\text{filter}} \) for a given query, an empty filtering set \( S_{\text{filter}} \), is initialized, and new route points are added which cannot be pruned using the existing points of a route in \( S_{\text{filter}} \). We organize all the filtered points in a point list, sorted by the number of routes which cover each point, and denote the route set and point set as \( S_{\text{filter}} \cdot R \) and \( S_{\text{filter}} \cdot P \) respectively, which are materialized using two dynamic sorted hash tables. Specifically, for \( S_{\text{filter}} \cdot R \), the key is the route ID, and the values are points of this route that cannot be filtered. For \( S_{\text{filter}} \cdot P \), the key is the route point ID, and the value is a list of routes containing the point.

Note that 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, and then this transition point can be pruned. We will employ this enhancement to achieve the more efficient pruning.

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

For example in Figure 3, \( R_1 \) and \( R_4 \) intersect at the second point \( R_{1,2} \), then \( C(R_{1,2}) = \{R_1, R_4\} \). Using the PLIST, we can retrieve the crossover route set of a point \( r \) easily, where \( C(r) = \text{PLIST}[r] \).

The crossover route set of each filter point \( r \) in \( S_{\text{filter}} \cdot P \) can be sorted by \( |C(r)| \) 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 2). 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 \( S_{\text{filter}} \), 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.

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

\[
\text{MinDist}(Q, c) = \min_{q \in Q} \text{MinDist}(q, c) \tag{3}
\]

In Line 12 of Algorithm 2, a point that cannot be pruned is a filter point and is added into \( S_{\text{filter}} \). First the route ID of the point is found, and inserted into \( S_{\text{filter}} \cdot R \). Then the point is inserted into the corresponding sort point list \( S_{\text{filter}} \cdot P \), and each point is affiliated with a list of route IDs containing it.

**Algorithm 2: FilterRoute(root\(_r\), Q, k)**

**Output:** \( S_{\text{filter}} \cdot S_{\text{refine}} \cdot S_{\text{result}} \)

1. minheap \( \leftarrow \emptyset \); \( S_{\text{filter}} \leftarrow \emptyset \); \( S_{\text{refine}} \leftarrow \emptyset \);  
2. minheap.push(root\(_r\));  
3. while minheap.isNotEmpty() do  
4. \( e \leftarrow \text{minheap.pop()} \);  
5. if \( e \) is a node then  
6. if isFiltered(\( Q, S_{\text{filter}} \cdot e \), \( k \)) then  
7. \( S_{\text{refine}} \text{.add}(e) \);
8. continue;  
9. if \( e \) is a point then  
10. \( S_{\text{filter}} \text{.add}(e, C(e)) \);
11. else  
12. foreach child \( c \) of \( e \) do  
13. minheap.push(\( c, \text{MinDist}(Q, c) \));  
14. return \( (S_{\text{filter}}, S_{\text{refine}}) \);  

**Algorithm 3: IsFiltered(Q, S_{\text{filter}}, node, k)**

**Output:** whether the node can be filtered

1. \( S \leftarrow \emptyset \);  
2. foreach \( p \in S_{\text{filter}} \cdot P \) do  
3. access list points in descending order  
4. if \( S \text{.size} > k \) then  
5. return true;  
6. label \( \leftarrow \) true;  
7. foreach \( q \in S_{\text{filter}} \cdot P \) do  
8. if located in \( H_{p,q} \) then  
9. label \( \leftarrow \) false;  
10. if label \( = \) true then  
11. \( S \leftarrow S \cup C(p) \); // crossover route set  
12. foreach route \( r \) \in \( S' \) do  
13. if \( S \text{.size} > k \) then  
14. return true;  
15. if VoronoiFiltering(Q, route) then  
16. \( S \leftarrow S \cup \{\text{route}\} \);  
17. return false;  

4.2.2 Transition Pruning

Based on the filter set \( S_{\text{filter}} \), 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 \( S_{\text{filter}} \) using Algorithm 3. Algorithm 4 uses the candidates in \( S_{\text{filter}} \) to check
whether $e$ is located in a filtering space of $Q$. The transition points that cannot be pruned are inserted into the candidate set for further refinement.

Algorithm 4 describes the procedure to prune the transition points using the generated filter set $S_{filter}$ from TR-tree. It is similar to the filtering method for generating the filtering set. The main difference with the traversal of Route R-tree is that only the un-pruned points need to be stored, and the filtering set $S_{filter}$ is fixed. As a result, a set of transition points $S_{end}$ is obtained which takes the query routes as $k$ nearest neighbors.

Algorithm 4: PruneTransition(root, $Q$, $S_{filter}$, $k$)

Output: $S_{end}$: candidate set
1. minheap ← $\varnothing$, $S_{end}$ ← $\varnothing$
2. minheap.push(root);
3. while minheap.isEmpty() do
   4. $e$ ← minheap.pop();
   5. if $e$ is a Node then
      6. if isFiltered($Q$, $S_{filter}$, $e$, $k$) then
         7. continue;
      8. if $e$ is a point then
         9. $S_{end}$-add($e$);
      else
         10. foreach child $c$ of $e$ do
             11. minheap.push($c$, $\text{MinDist}(c, Q)$);
   12. return $S_{end}$;

4.2.3 Verification

The verification mainly uses the filtered node set $S_{refine}$ during the traversal of RR-tree to find $S_{filter}$ in Algorithm 3. It can be divided into two steps. First, the nodes in RR-tree encountered during the filtering phase are kept in $S_{refine}$ in Line 7 of Algorithm 3. The verification algorithm runs in rounds. In each round, one of the nodes in $S_{refine}$ is opened and its children are inserted into $S_{refine}$. During each round, the nodes and points in $S_{refine}$ are used to identify the candidates that can be verified using $S_{refine}$, which are the nodes confirmed as $R_{k}$:NNT or guaranteed not to be $R_{k}$:NNT. Such candidates are verified and removed from $S_{end}$. The algorithm terminates when $S_{end}$ 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 $S_{refine}$ which are closer to the query than the candidates, then it can safely be removed from $S_{end}$. Hence, we maintain a set to store the unique IDs of these routes when every candidate point is checked in $S_{end}$. The route IDs are found, and the set is updated using NList when new filtering points or nodes from $S_{refine}$ are found. When the number of IDs in a set is greater than $k$, it can be removed from $S_{end}$.

After finding the transition points for the routes, as they will take the query as $k$ nearest neighbors, then for the $R_{k}$:NNT, the transition ID for all remaining points can be returned as the final result $S_{result}$ in the second step. While $V_{k}$: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 $S_{result}$.

5. OPTIMIZATIONS ON 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, $S_{filter}$ can be used for additional pruning. Given a query and a filtering route $R$, 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 called as the Voronoi cell, and the point is called the kernel of this cell. By plotting the Voronoi diagram $V_{k}Q$ between the query $Q$ and a filtering route $R$, as shown in Figure 5, The Voronoi cell $V_{k}Q[p]$ of the route $R$ can be found, and any point inside these cells is 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 $R$ and query $Q$, we define the Voronoi filtering space as:

$$H_{R,Q} = \bigcup_{p \in R} V_{k}Q[p]$$

which is a union of the Voronoi cells of all points from $R$, and $V_{k}Q$ is the Voronoi diagram of union of points from $R$ and $Q$.

Any transition point inside $H_{R,Q}$ 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 can both find a point in the filtering route rather than the query, so the transition point will not the take query as the nearest neighbor.

The filtering route $R$ is used to further prune the transition point if it cannot be pruned by the filter points in $R$ one by one as shown in Line 7-10 of Algorithm 3. After scanning all the filtering points of a route in $S_{filter}$, $F$, we will use the Voronoi filtering space of the query 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 one more 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

Figure 5: Pruning based on the Voronoi diagram of a query (red) and a filtering route composed of 4 points (black).
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 of the proposed method will be complex 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 length is larger. To alleviate this problem, we introduce a divide-and-conquer method based on our processing framework.

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

$$R_k NNT(Q) = \bigcup_{q_i \in Q} R_k NNT(q_i)$$  \hspace{1cm} (5)

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

Based on this observation, a divide and conquer framework is proposed that uses multiple $R_k NNT$ searches which were introduced in Section 4. The main idea is that $R_k NNT$ 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 3 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 search 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 how 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 high possibility to get high profit by attracting bus passengers to take the Uber to travel alternatively. Next, we will introduce the maximizing $R_k NNT$ in the bus network.

Here we take 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 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

![Figure 6: Frequency histogram of the ratio between travel distance and straight-line distance for all routes in LA and NYC.](image)

$R = (v_1, v_2, \ldots, v_n) \in V \times V \times \cdots \times V$ 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 respectively.

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

$$\psi(R) = \sum_{p_i \in R & i \in [1,n-1]} |\text{distance}(p_i, p_{i+1})|$$  \hspace{1cm} (6)

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

**Definition 10.** ($MaxR_k NNT$) Given a distance threshold $\tau$, a starting vertex $v_s$ and a destination vertex $v_e$, $MaxR_k NNT(v_s, v_e, \tau)$ returns an optimal route $R$ from $S_{se}$ such that $\forall R' \in S_{se} - R$ we have $|\omega(R')| \leq |\omega(R)|$ 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 $|\omega(R')| \leq |\omega(R)|$ to $|\omega(R')| \geq |\omega(R)|$ 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.

**Baseline.** A brute force method for $MaxR_k NNT$ is to find all the candidate routes which meet the travel distance threshold constraint. This can be done by extending the $k$ shortest path method proposed by [23][12] with a loop to find the sub-optimal route until the distance threshold $\tau$ 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 1 where almost all routes such as $\text{abej, acej}$ and $\text{acdej}$ 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 7.5. For the 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 will not work well. 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 $\mathcal{G}$.

6.2 Our Solution

According to Lemma 3 the query $Q$ can be decomposed into a set of $|Q|$ queries, which means that we can get the pre-computed $R_k NNT$ set for every vertex, and conduct 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 $G$, an $R$\textsuperscript{kNN} query is ran, and the result stored. A pre-computed matrix $M[i][j]$ is created which stores the pre-computed all-pair shortest distance for all vertex pairs in $G$ using the Floyd-Warshall algorithm $[4]$. The details of pre-computation can be found in Algorithm $[5]$.

Algorithm 5: Precomputation($G, D_{R_1}, D_{T_1}, k$)

Output: $G.V$: the vertexes with $R$\textsuperscript{kNN} set
1. root\textsubscript{i} $\leftarrow$ CreateIndex($D_{T_1}$);  // root of $R$-tree
2. root\textsubscript{j} $\leftarrow$ CreateIndex($D_{R_1}$);  // root of $T$-tree
3. foreach vertex $v$ $\in G$ do
4. $S_{result} \leftarrow R$\textsuperscript{kNN}($v$, root\textsubscript{i}, root\textsubscript{j});  // call Algorithm [5] by query $v$
5. $G.V.R$\textsuperscript{kNN}($v$) $\leftarrow S_{result}$;  // update the set on vertex
6. foreach vertex $v'$ $\in G - v$ do
7. $M[v][v'] \leftarrow$ ShortestDistance($G, v, v'$);
8. return $G.V$;

With the pre-computed $R$\textsuperscript{kNN} 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 $R$\textsuperscript{kNN} set of each route can be found by performing a union operation on the sets. Compared with the baseline method, the on-the-fly $R$\textsuperscript{kNN} query is replaced with pre-computation, and the running time for the search 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 2.** As shown in Figure [7], the red points are the start $O = a$ and end $D$ respectively. A query formed by these two points and $\tau = 6$ return the route with largest $R$\textsuperscript{kNN} 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 $R$\textsuperscript{kNN} set for each vertex. So, $\omega(af.hd) = \{T_1, T_2, T_3, T_4, T_5\}$ and $\mu(\omega(af.hd)) = 1.5 + 1.4 + 1.5 = 4.5$.

6.2.2 Route Searching by Pruning

After getting the $R$\textsuperscript{kNN} set for every vertex in the graph $G$, Algorithm [5] can be run 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 $Q$, and the partial route is increased until it meets the end point $v_e$ and has the maximum result set size. Specifically, the two checking functions work as below:

Algorithm 6: MaxR\textsuperscript{kNN}($o, d, \tau$)

Output: $R$: the optimal route
1. if checkReachability($v_i, v_e, \tau$) then
2.  return $\emptyset$;
3.  $R \leftarrow \emptyset, R^* \leftarrow \{v_i\}$;  // travel distance
4.  $\omega(R^*) \leftarrow 0$;  // $\omega$ is created which
5.  $\omega(R^*) \leftarrow G.V.R$\textsuperscript{kNN}($v_i$);  // $R$\textsuperscript{kNN} set
6.  $Q \leftarrow \emptyset$;  // queue stores the partial routes
7.  push($Q, \{R^*, \psi(R^*), \omega(R^*)\}$);
8.  while $Q \neq \emptyset$ do
9.     $\{R^*, \psi(R^*), \omega(R^*)\} \leftarrow$ pop($Q$);
10.    $v_i \leftarrow$ GetEnd($R^*$);
11.    foreach $v_j \in$ Neighbor($G, v_i$) do
12.       if checkReachability($v_j, v_e, \tau - \psi(R^*)$) then
13.           if checkDominance($a, v_j, v_e, \psi(R^*), \omega(R^*)$) then
14.               $S \leftarrow$ Update($D_T[d], (\psi(R^*), \omega(R^*))$);
15.               foreach candidate $\in S$ do
16.                   Delete($Q$, candidate);
17.                   $R^* \leftarrow R^* \cup \{v_j\}$;
18.                   $\psi(R^*) \leftarrow \psi(R^*) + \psi(v_i, v_j)$;
19.                   $\omega(R^*) \leftarrow \omega(R^*) + \omega(R^*)$;
20.                   push($Q, \{R^*, \psi(R^*), \omega(R^*)\}$);
21.           if GetEnd($R^*) = v_e$ then
22.               if $\omega(R^*) > \omega$ then
23.                   $R \leftarrow R^*$;
24.                   $\omega \leftarrow \omega(R^*)$;
25.       while $Q \neq \emptyset$ do
26.           return $R$;

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 $\omega(af.hd) > \tau - \psi(R^*)$, it will return false and move to next neighbor of vertex $v_i$ in $G$.

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 short route and a larger $R$\textsuperscript{kNN} set, then it can dominate the current route. Specifically, a dominating lemma is introduced which works for both $\forall R$\textsuperscript{kNN} and $\exists R$\textsuperscript{kNN}.

**Lemma 4.** Given two partial routes $R^*_1$ and $R^*_2$ which have the same start and end, $R^*_2$ dominates $R^*_1$ in $MinR$\textsuperscript{kNN} ($R^*_2$ dominates $R^*_1$ in $MinR$\textsuperscript{kNN} when $|\psi(R^*_2)| < |\psi(R^*_1)|$ and $|\omega$\textsuperscript{NN}(R$^*_2$)| > $|\omega$\textsuperscript{NN}(R$^*_1$)|).

**Proof.** For purposes of proving the lemma, we use $\omega(R)$ and $\omega^*$ ($R$) to represent $\forall R$\textsuperscript{kNN} ($R$) and $\exists R$\textsuperscript{kNN} ($R$) to distinguish them. Given any partial route $R$ which starts at $v_i$ and ends at $d$, $R^*_1$ and $R^*_2$ can be connected to form two complete routes $R_1$ and $R_2$. 1) For $\exists R$\textsuperscript{kNN}, if $|\omega^*(R^*_1)| > |\omega^*(R^*_2)|$, then $|\omega(R_1)| \geq |\omega^*(R_1)|$, as there is no intersection between $\omega^*(R^*_1)$ and $\omega^*(R^*_2)$ because $T \in \omega^*(R^*_1)$ is the set of transitions that have $k$\textsuperscript{NN} in $R^*_1$ for both origin and destination points. Given

![Figure 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\textsuperscript{kNN} set for each vertex.](image-url)
that $|\omega(R_2)| \leq |\omega(R_2^*)| + |\omega(R_2')|$, while $\psi(R_1^*) < \psi(R_2^*)$, 2) $\forall R\in\mathbb{R}^N$, $|\omega^*(R_2^*)| \leq |\omega(R_2^*)| + |\omega^*(R)|$, 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 2, a dynamic table $DT$ 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 $R\in\mathbb{R}^N$ 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 $DT$ that are dominated by the new route $R^*$ are removed. If a new one is found that dominates $R^*$, the loop terminates, and the next partial route is processed.

Example 3. In Figure 2, $\{a, b, 20\}$ is added to the queue $Q_0$ after checking the reachability from $a$ to $b$ by comparing the pre-computed shortest distance with $\tau$. Then, pop the queue $Q_0$ 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 $\psi(\overrightarrow{be}) = 3.8 < (6 - 1.6)$, so $\{a, b, 1.6, \{T_1, T_2\}\}$ is added to $Q$. Similarly, $\{a, c\}$ is inserted into $G$. $\{a, c\}$ cannot be enqueued as the shortest distance from $a$ to $j$ by $\psi(\overrightarrow{afj}) = 5.2 > (6 - 1)$. $\{a, b, e, 3.1, \{T_1, T_2\}\}$ and $\{a, c, e, 2.6, \{T_1, T_2, T_3, T_4\}\}$ are enqueued and $DT[\{e\}] = \{a, b, e, 3.1, \{T_1, T_2\}\}$ is updated. Further, $\{a, c, e, f, 4.5, \{T_1, T_2, T_3, T_4\}\}$ is enqueued. $\{a, b, e, h, 4.5, \{T_1, T_2\}\}$ has a greater travel distance, and $\omega(\overrightarrow{ache}) = \{T_1, T_2\}$, and $\omega^*(\overrightarrow{acfh}) = \{T_1, T_2, T_3, T_4\}$, so $\omega^*(\overrightarrow{acfh}) > \omega(\overrightarrow{ache})$, $\overrightarrow{acfh}$ dominates $\overrightarrow{ache}$. Based on this extension in the graph, when $Q_0$ is empty, the algorithm terminates.

For Min$\mathbb{R}^N$, line 8 is changed to max $\leftarrow \infty$, and line 2 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 Experimental Setup

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

| Dataset | $|\mathcal{D}_R|$ | $|G, E|$ | $|G, V|$ |
|---------|----------------|----------|----------|
| LA-Route | 1,208 | 72,346 | 14,119 |
| NYC-Route | 2,022 | 61,118 | 16,999 |

Table 3: Transition Datasets.

| Dataset | $|\mathcal{D}_T|$ | Latitude | Longitude |
|---------|----------------|----------|----------|
| LA-Transit | 109,036 | [32°35’] | [−120°11’] |
| NYC-Transit | 195,833 | [39°42’] | [−75°17’] |
| NYC-Synthetic | 10,000,000 | [39°42’] | [−75°17’] |

Table 4: Parameter Settings.

We conducted experiments to evaluate our solutions to Min$\mathbb{R}^N$ and Max$\mathbb{R}^N$ using real bus route data and check-in data from Foursquare in New York and Los Angeles, which are two largest cities in the USA. We have published our dataset 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.

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 and Los Angeles. 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 7 and statistics for the cleaned data is shown in Table 2. 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$\mathbb{R}^N$ 7

Algorithms for evaluation. We compared the following methods when processing R$\mathbb{R}^N$ over the two datasets. (1) Filter-Refine: The basic framework proposed in Section 4. (2) Voronoi: 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) Divide-Conquer: As 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 $\mathcal{D}_R$. 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. All experimental results are averaged by running all 1,000 queries. 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.

1https://foursquare.com/
2https://sites.google.com/site/shengwangcs/home/rknn
3http://web.mta.info/developers/developer-data-terms.html#data
4http://developer.metro.net/gtfs/google_transit.zip
Parameters. Table 1 summarizes all key parameters for a query, and the default values are underlined. $I = \psi(Q)$ is the interval length between two adjacent points in the query, where $\psi(Q)$ is the travel distance of the query route and can be computed by Equation 6.

Figure 9: Effect on Running Time with the increasing of $k$.

Figure 10: Breakdown of running time with increasing $k$ in LA.

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

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. We can see that the verification occupies more than 80% for most cases.

Effect of $k$. Figure 9 shows that the time cost for all three methods will increase 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 $I$. We observe that the intervals $I$ 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 $I$ 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.

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 14 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.

In summary, our main observations are:

1. Divide-Conquer consistently has the best performance, followed by Voronoi, with Filter-Refine being the worst.
2. All three methods are sensitive to $k$ and $|Q|$. Only Filter-Refine and Voronoi are sensitive to the interval length $I$ of the query.
3. When taking existing routes as real queries, most queries can be answered efficiently using Divide-Conquer.

7.3 Evaluation of MaxR/kNNT

Algorithms for evaluation. (1) BruteForce: the baseline method which uses the $k$ shortest paths [23] to find all the routes which have a smaller travel distance than the distance threshold $\tau$, after which an R/kNNT is performed on every candidate to choose the maximal one. (2) Pre: the method that extends the BruteForce by pre-computation of the R/kNNT set for every vertex without an on-the-fly R/kNNT query. For MaxR/kNNT and MinR/kNNT, 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 MaxR/kNNT and MinR/kNNT search algorithm to find the new “optimal” routes. Finally, we compared the R/kNNT sets of the original routes against the new routes.

Parameters. We discovered two key parameters that affect the performance of MaxR/kNNT: (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) $\frac{\psi(Q)}{\psi(se)}$, 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 5 shows the time spent on pre-computation. The pre-computation consists of of two steps: the R\&NNT 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 R\&NNT search of all vertices in $G$ costs less than 5 minutes when $k = 10$. For the synthetic dataset which contains 10M transitions, 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 Brute-force, the reasons are twofold: (1) It returns more candidate routes for R\&NNT; (2) The candidate routes are longer when $\psi(se)$ is long, so more time has to be spent for every R\&NNT query. In contrast, for the remaining three methods, since we have pre-computed the R\&NNT set for every vertex, the running time comes from the search over $G$. Pre-Max has the best performance due to the bound checking during the spreading of partial routes.

Effect of $\frac{\psi(se)}{|Q|}$. To generate the query, we choose a subset of queries with a fixed $\psi(se)$ as the default value shown in Table 3 and alter $\tau$ in the experiment. Figure 19 shows that increasing $\frac{\psi(se)}{|Q|}$ leads to an increased running time. The reason can also be ascribed to the increasing number of candidates between the origin and destination.

Real queries. We took each route in $D_{re}$ as a query to perform an MaxR\&NNT search to see whether we can find a better route which has a larger R\&NNT 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. We can see that most 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\&NNT route which attracts the most passengers, 4) the MinR\&NNT route which attracts the fewest passengers. The right table shows the search time, number of passengers, travel distance, and number of stops for these four routes. We find: (1) the original route and the MaxR\&NNT route are almost the same (in particular, MaxR\&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 locations. (2) If a driver wants to save time, the least crowded route can be selected as provided by MinR\&NNT; if the car should be shared to increase revenue, the route found by MaxR\&NNT is a good choice.

8. CONCLUSION

In this paper, we proposed and studied the R\&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\&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.

9. REFERENCES

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

[2] 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.

[3] 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.

[4] C. Burfield. Floyd-Warshall Algorithm All-Pairs Shortest Paths. pages 1–15, 2013.

[5] A. Ceder. Public Transit Planning and Operation. 2007.

[6] M. A. Cheema, X. Lin, W. Zhang, and Y. Zhang. Influence zone: Efficiently processing reverse k nearest neighbors queries. Proceedings - International Conference on Data Engineering, pages 577–588, 2011.

[7] 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.

[8] 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.

[9] T. Enrich, H.-P. Kriegel, N. Manouilis, J. Niedermayer, M. Renz, and A. Zäffle. Reverse-nearest neighbor queries on uncertain moving object trajectories. In Database Systems for Advanced Applications, pages 92–107. Springer, 2014.

[10] Y. Liu, C. Liu, J. M. Yuan, L. Duan, Y. Fu, H. Xiong, S. Xue, and J. Wu. Intelligent bus routing with heterogeneous human mobility patterns. Springer London, 2016.

[11] 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.

[12] E. Q. V. Martins and M. M. B. Pascoal. A new implementation of Yen’s ranking loopless paths algorithm. 4or, 1(2):121–133, 2003.

[13] S. Pattanaik, S. Mohan, and V. Tom. Urban bus transit route network design using genetic algorithm. Journal of transportation engineering, 124(4):368–375, 1998.

[14] 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 19th ACM SIGSPATIAL International Conference on

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

Figure 19: Effect on running time with the increase of $\tau/\psi$.

Figure 20: Distribution of running time of MaxRkNNT on real route query.

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

Advances in Geographic Information Systems, ACM SIGSPATIAL GIS 2011, pages 181–190.

[15] I. Stanoi, D. Agrawal, and A. El 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.

[16] Y. Tao, D. Papadias, and X. LIan. Reverse kNN search in arbitrary dimensionality. In Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, pages 744–755. VLDB Endowment, 2004.

[17] R. C.-w. Wong, M. T. Oszu, P. S. Yu, A. W.-c. Fu, and L. Liu. Efficient Method for Maximizing Bichromatic Reverse Nearest Neighbor. Vldb, 2(1):112–1137, 2009.

[18] W. Wu, F. Yang, C.-Y. Chan, and K.-L. Tan. Finch: Evaluating reverse k-nearest-neighbor queries on location data. Proceedings of the VLDB Endowment, 1(1):1056–1067, 2008.

[19] O. Y. Xian, M. Chitre, and D. Rus. An approximate bus route planning algorithm. Proceedings of the 2013 IEEE Symposium on Computational Intelligence in Vehicles and Transportation Systems, CIVTS 2013 - 2013 IEEE Symposium Series on Computational Intelligence, SSCI 2013, pages 16–24, 2013.

[20] S. Yang, M. A. Cheema, X. Lin, and W. Wang. Reverse k nearest neighbors query processing: experiments and analysis. Proceedings of the VLDB Endowment, 8(5):608–616, 2015.

[21] S. Yang, M. A. Cheema, X. Lin, and Y. Zhang. SLICE: Reviving regions-based pruning for reverse k nearest neighbors queries. Proceedings - International Conference on Data Engineering, 1:760–771, 2014.

[22] Z. Yang, B. Yu, and C. Cheng. A parallel ant colony algorithm for bus network optimization. Computer-Aided Civil and Infrastructure Engineering, 22(1):44–55, 2007.

[23] J. Y. Yen. Finding the K Shortest Loopless Paths in a Network. Management Science, 17(11):712–716, 1971.

[24] Z. Zhou, W. Wu, X. Li, M. L. Lee, and W. Hsu. MaxFirst for MaxBRkNN. Proceedings - International Conference on Data Engineering, pages 828–839, 2011.