Discovering Real-time Reachable Area using Trajectory Connections

Ruiyuan Li[1,2], Jie Bao[2], Huajun He[3,2], Sijie Ruan[1,2], Tianfu He[4,2], Liang Hong[5], Zhongyuan Jiang[1], and Yu Zheng[1,2⋆]

[1] Xidian University, Xi'an, China
liruiyuan@whu.edu.cn, sjruan94@gmail.com, zyjiang@xidian.edu.cn
[2] JD Intelligent City Research, Beijing, China
baojie@jd.com, msyuzheng@outlook.com
[3] Southwest Jiaotong University, Chengdu, China
hehuajun@my.swjtu.edu.cn
[4] Harbin Institute of Technology, Harbin, China
Tianfu.D.He@outlook.com
[5] Wuhan University, Wuhan, China
hong@whu.edu.cn

Abstract. Discovering real-time reachable areas of a specified location is of importance for many location-based applications. The real-time reachable area of a given location changes with different environments. Existing methods fail to capture real-time traffic conditions instantly. This paper provides the first attempt to discover real-time reachable areas with real-time trajectories. To address the data sparsity issue raised by the limited real-time trajectories, we propose a trajectory connection technique, which connects sub-trajectories passing the same location. Specifically, we propose a framework that combines indexing and machine learning techniques: 1) we propose a set of indexing and query processing techniques to efficiently find reachable areas with an arbitrary number of trajectory connections; 2) we propose to predict the best number of connections in any location and at any time based on multiple datasets. Extensive experiments and one case study demonstrate the effectiveness and efficiency of our methods.

1 Introduction

Real-time reachable area discovery aims to find the reachable area from a specified location within a given time period in real-time conditions. It is very useful in many urban applications: 1) Location-based recommendation. As depicted in Fig. 1(a), a user wants to find the restaurants that can be reached from her current location within 5 minutes; and 2) Vehicle dispatching. As illustrated in Fig. 1(b), a user calls for a taxi to pick her up in 10 minutes. Taxi companies would use this function to find the candidate drivers. Traditional methods are based on the static spatial range query over either Euclidean distance [1] or road network distance [2, 3], which find the same reachable areas without considering the highly skewed traffic conditions at different time (e.g., late night vs. rush hours). Optional methods first estimate the travel time of each road segment [4–7], then find reachable areas using road network expansion techniques [2, 8].

⋆ Yu Zheng and Jie bao are the corresponding authors of this paper.

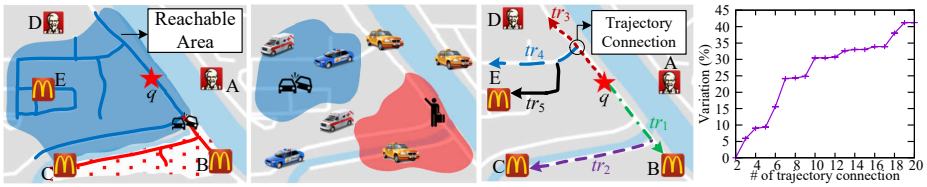


Fig. 1. Application Scenarios and Trajectory Connections.

However, these methods ignore the delays of intersections. Besides, they are designed to model the regular traffic conditions, but can hardly capture abnormal events, such as accidents. With the availability of massive trajectories, [9] takes advantage of historical trajectories that passed the query location during the request hour to find the reachable area. However, this approach cannot be applied directly in a real-time scenario, as it does not consider real-time contexts, such as weather, traffic conditions, accidents and other events in a city.

An intuitive idea is to use only real-time trajectories (e.g., generated within the most recent one hour). However, we cannot apply directly the techniques in [9] to real-time trajectories, due to the **data sparsity** issue (i.e., the number of trajectories passing the query location in a short time window is very limited). To solve this issue, we propose a **trajectory connection** technique. As illustrated in Fig. 1(c), if we consider only the trajectories that exactly pass the query location q, i.e., tr_1 and tr_3, only B can be reached. Suppose the trajectories can be connected if they share the same locations, e.g., tr_1 and tr_2, C is also in the reachable area. Further, if the trajectories can be connected twice, E can be reached as well by connecting tr_5 to tr_4, which significantly improves the coverage of the reachable area. However, the reliability of discovered reachable areas may be affected by trajectory connections, as the connected trajectories are generated by different moving objects, where the time cost of connections (e.g., waiting time in crossroads) is ignored. To study the effects of trajectory connections on reliability, we compare the estimated travel time of a path using different numbers of trajectory connections with the real travel time, as shown in Fig. 1(d). It shows that, with more connections, the accuracy of estimation becomes lower. But if we limit the number less than five, the estimation variation is less than 10%, which guarantees a reasonable reachable area. However, a small trajectory connection number may cause a coverage problem. As a result, the number of trajectory connections is a trade-off between reliability and coverage.

An appropriate connection number is determined by the real-time trajectories. If there are fewer real-time trajectories, a bigger connection number should be assigned to achieve a good coverage. However, it is hard to determine a good connection number, as the spatio-temporal distribution of trajectories is skewed severely. For example, downtown areas contain more taxi activities than suburb areas. Meanwhile, there are usually more taxi activities during rush hours. Therefore, a **dynamic** connection number is needed when a query arrives.

There are three main challenges. 1) As each trajectory can be connected at any location with numerous trajectories, it results in exponential numbers of possible combinations, which can be prohibitively inefficient. 2) A good connection

number is determined by the real-time trajectories, which is further affected by multiple complex factors, e.g. weather conditions, road networks, and land usage around the request location [10]. 3) there is even no ground truth of reachable areas in our datasets, which leads to lack of the labels of connection numbers for model learning. The main contributions of this paper are summarized as follows:

(1) We provide the first attempt to discover real-time reachable areas with dynamic trajectory connections, and design a framework that combines indexing with machine learning techniques to solve this problem (Sect. 3).

(2) We design a set of indexing and query processing techniques to prune redundant trajectory connections, which can efficiently answer real-time reachable area discovery requests with arbitrary connection numbers (Sect. 4).

(3) We propose a method to generate the labels of connection numbers using historical trajectories, and identify spatio-temporal features to predict a good connection number in any location and at any time (Sect. 5).

(4) Extensive experiments are conducted using multiple real datasets, verifying the effectiveness and efficiency of our solutions. Readers can experience our demo system in http://r-area.urban-computing.com/. (Sect. 6)

2 Preliminary

Definition 1 (Road Network) *A road network RN is a directed graph* $G = (V, E)$*, where* $V = \{v_1, v_2, ..., v_m\}$ *is a set of vertices representing the intersections, and* $E = \{e_1, e_2, ..., e_n\}$ *is a set of road segments (edges) with directions.* $e.v_{start}$ *and* $e.v_{end}$ *represent the start vertex and end vertex of edge e respectively.*

Definition 2 (Map-Matched Trajectory) *A map-matched trajectory* $tr =< (e_1, t_1) \rightarrow (e_2, t_2) \rightarrow ... \rightarrow (e_n, t_n) >$ *is generated by mapping raw GPS points onto the corresponding road segments, where* t_i *is the time when the trajectory enters edge* e_i*. The time cost to traverse* e_i *is* $Cost(tr.e_i) = t_{i+1} - t_i$*.*

For simplicity, in this paper, we represent a map-matched trajectory without detailed temporal information, i.e., $tr =< e_1 \rightarrow e_2 \rightarrow ... \rightarrow e_n >$. $tr[i...j]$ denotes the sub-trajectory of tr that starts from i-th edge to j-th edge in tr.

Definition 3 (Connected Trajectory). *A connected trajectory* $ctr =< tr_1[i_1...j_1] \rightarrow tr_2[i_2...j_2] \rightarrow ... \rightarrow tr_n[i_n...j_n] >$ *consists of a sequence of sub-trajectories, where the last edge of the previous sub-trajectory shares the same intersection with the first edge of the next sub-trajectory, i.e.,* $tr_m[j_m].v_{end} = tr_{m+1}[i_{m+1}].v_{start}$*.*

The number of sub-trajectories in a connected trajectory ctr is its ***degree***, denoted by $D(ctr)$. Specifically, there is $D(ctr) - 1$ connections in ctr.

Problem Definition. Given a real-time trajectory database $\mathcal{T}$ generated in the most recent time δ, a query location q, a time budget t, and external environmental data around q (e.g. POIs, road networks, and meteorological data), we first predict a reasonable degree constraint $k \geq 1$ of q, and then find a set of road segments as the reachable area $RA(\mathcal{T}, q, t, k)$, such that for any $e_i \in RA$, there exists at least one connected trajectory $ctr =< q \rightarrow ... \rightarrow e_i >$ that connects e_i from q, satisfying the following two constraints:

(1) **Time Constraint.** The time cost of ctr is not greater than t:

$$Cost(ctr) = \sum_{m=1}^{D(ctr)} Cost(tr_m[i_m, j_m]) \leq t \tag{1}$$

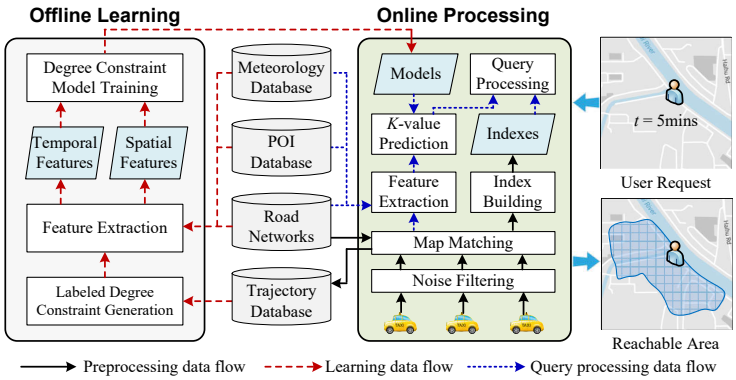


Fig. 2. System Framework.

(2) **Degree Constraint.** The degree of ctr is not greater than k:

$$D(ctr) \leq k \tag{2}$$

The degree constraint k defines the maximum number of sub-trajectories in a connected trajectory, which provides a trade-off between the coverage and reliability of reachable areas. To guarantee a high reliability, we set $1 \leq k \leq 5$ according to Fig. 1(d). Besides, we focus on reachable area discovery in a very short time ahead, e.g., $t \leq 30$ minutes, as it can satisfy most dispatching or emergency scenarios. We also have $\delta \times k \geq t$, to make the connection feasible.

3 Framework

Figure 2 gives the framework with two major parts, offline learning and online processing, which generates three data flows:

Preprocessing data flow. This data flow (black solid arrows) takes real-time GPS updates as input, removes the trajectories with abnormal speed, and maps the GPS points onto their corresponding road segments. The map-matched trajectories are then stored in a trajectory database for offline learning, and used for online index building and degree constraint prediction. We adopt the techniques in [11–14] to process the trajectory data based on our system JUST [15, 16].

Learning data flow. In this flow (red broken arrows), we first generate the labels of degree constraints, then extract features from various datasets. Finally, these features are leveraged for training models, with which the best degree constraint in any location and at any time can be predicted.

Query processing data flow. In this data flow (dotted blue arrows), when a user request arrives, we first extract the spatio-temporal features in the given location from multiple data sources, then predict the best degree constraint with the models trained offline. Finally, the real-time reachable area is calculated by means of the built indexes and predicted degree constraint.

As we will apply the indexing techniques to degree constraint model learning, we first introduce the index building and query processing techniques in Sect. 4, and then detail degree constraint model training and prediction in Sect. 5.

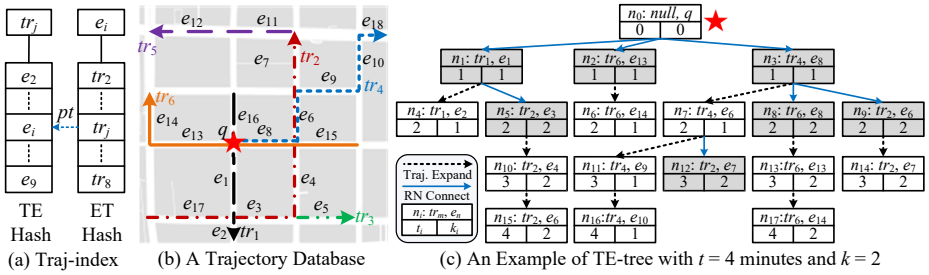


Fig. 3. Example of Traj-index and TE-tree.

4 Index Building & Query Processing

In this section, we assume the degree constraint k is already predicted. If we apply the traditional network expansion based algorithm [2, 8] directly, in each expansion step, each candidate road segment is associated with a status of two different dimensions, i.e., time cost t_c and degree cost k_c, which makes it impossible to select the "best" candidate. Therefore, it is required to build an effective index and an efficient pruning strategy to discover real-time reachable areas.

4.1 Traj-index

Data Structure. *Traj-index* builds links between edges and trajectories. Figure 3(a) gives an example with two parts: 1) *Trajectory-Edge (TE) hash* uses trajectory IDs as hash keys, and each value is a list of edge IDs passed by the trajectory within the most recent δ minutes; 2) *Edge-Trajectory (ET) hash* is an inverted index where the keys are edge IDs, and each value is a list of trajectory IDs passing the edge ordered by arriving time within the most recent δ minutes. To efficiently expand the search via sub-trajectories, a pointer is maintained to link the same trajectory-edge combination between the two hash tables.

Construction. *Traj-index* is updated in a streaming way, where each update is processed incrementally. The complexity is $\mathcal{O}(m \times n)$, where m is the number of new trajectories, and n is the average size of each trajectory. As a result, it is efficient to handle large-scale trajectory updates in a real-time manner.

Query Processing. With *Traj-index*, we propose a query processing method *trajectory expansion* based on an intuitive idea: 1) traversing all trajectories passing q, and finding covered road segments; 2) for each qualified road segment, identifying all possible trajectory connections, and updating new qualified road segments; and 3) repeating the previous step, until the budget t or k is used up.

To realize the discovery of reachable areas with *Traj-index*, a *TE-tree* is created during the search process. For example, given a trajectory database as Fig. 3(b), we get a *TE-tree* shown as Fig. 3(c), where the query location q forms the root. The *TE-tree* consists of one type of nodes and two types of links: 1) ***TE-node.*** Each node contains five properties: an identifier n, a trajectory tr, an edge e, a time cost t_c, and a degree cost k_c. A *TE-node* indicates the current search status (i.e., trajectory tr at edge e), where the time cost t_c and degree cost k_c are the corresponding costs traveling from the root. 2) ***Expansion Link.*** This link (denoted as the dotted black arrows) is generated by accessing the *TE hash* in *Traj-index*. The nodes, e.g., $n_i \& n_j$, along a link belong to the same

trajectory, with an increasing time cost t_c and the same degree cost k_c, i.e., $n_j.tr = n_i.tr$, $n_j.t_c = n_i.t_c + Cost(n_j.e)$, $n_j.k_c = n_i.k_c$. 3) ***Connection Link.*** This link (denoted as the blue solid arrows) is generated by the connection of different two sub-trajectories, which can be built efficiently with road networks and the *ET hash*. The nodes, e.g., $n_i \& n_j$, connected by this type of link have an increasing time cost t_c and an increasing degree cost k_c, i.e. $n_j.tr \neq n_i.tr$, $n_j.t_c = n_i.t_c + Cost(n_j.e)$, $n_j.k_c = n_i.k_c + 1$.

Note that *TE-tree* is constructed during the search process, which cannot be pre-computed. As we enumerate all possibly connected trajectories for each candidate edge via the nodes in *TE-tree*, finding a reachable area of a position can be reduced to traversing its corresponding *TE-tree*. An intuitive idea uses a depth-first approach, until the budget t or k are used up (denoted as **TE**). However, there will be many edges being visited redundantly. For example in Fig. 3(c), n_2 and n_{13} are traversed with the same trajectory tr_6 and edge e_{13}. As shown in Fig. 3(b), there is an illogical path combination: the user first goes right with tr_4 on e_8, and then makes a U-turn and goes back to e_{13}. A more reasonable route should go directly to e_{13}, which is represented as n_2 in *TE-tree*.

To avoid redundant computation, we design a pruning strategy based on the observation that, illogical routes always start by a *TE-node* with the same trajectory and edge of some visited nodes, but with higher time and degree costs than them, e.g., n_{13} and n_2 in Fig. 3(c). We call this as *node domination*.

Definition 4 (Node Domination in TE-tree) *Given two nodes n_i and n_j in TE-tree, if $n_i.tr = n_j.tr$, $n_i.e = n_j.e$, $n_i.t_c \leq n_j.t_c$, and $n_i.k_c \leq n_j.k_c$, then n_i dominates n_j, denoted as $n_i \succeq n_j$.*

Theorem 1 *If $n_i \succeq n_j$, n_j and all the children of n_j can be pruned.*

Proof. As $n_i \succeq n_j$, both nodes have the same trajectory and edge, all possible connected trajectories from n_j (which generate the children nodes of n_j in *TE-tree*) can also be attached to n_i. As a result, all edges covered in n_j's children are also covered in n_i's children. Thus, we can safely prune n_j and all of its children.

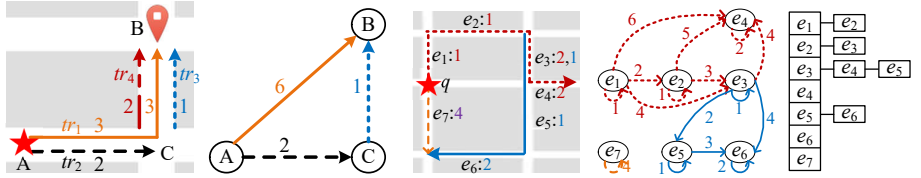
To maximize the pruning ability, it is important to apply a good order to traverse *TE-tree*. For example, in Fig. 3(c), n_{13} and its children can be pruned only if n_2 is visited before n_{13}. As a result, the nodes with smaller t_c and k_c should be searched as early as possible. We propose two heuristics: 1) *H1:* Nodes with the same trajectory are searched in priority, as it guarantees not to increase the degree cost; 2) *H2:* For multiple sub-trajectories connecting to the same node, we search the trajectory with the lowest time cost first.

The proposed method **TE+** (Algorithm 1) with the two heuristics starts from the root of *TE-tree*, and performs a k-iteration process, where each iteration has two steps: 1) *Connection*, which connects the existing *TE-nodes* with possible road segments based on the road network adjacency. Each connection consumes one degree budget. 2) *Expansion*, which generates new *TE-nodes* by expanding trajectories from the newly added road segments. To ensure *H2*, we resort to a priority queue to store all candidate *TE-nodes* based on their time costs. We also record all visited *TE-nodes* with a set. If there exists a visited node dominating the newly generated node n, we prune n according to Theorem 1.

Algorithm 1: TE+

Input: *Traj-index* of $\mathcal{T}$, query location q, time constraint t, degree constraint k
Output: Reachable area $RA(\mathcal{T}, q, t, k)$.

```
1  Initialize a queue ConQueue to record the candidate connection node;
2  Initialize a set Visited to record all visited TE-nodes;
3  Form the root of TE-tree with q, and add it to ConQueue;
4  for i = 1 to k do
5      Init a empty priority queue pq;
       // Connection Step
6      Pop all nodes in ConQueue, create new nodes based on the road network
         adjacency and Edge-Trajectory Hash, and add the new nodes to pq;
       // Expansion Step
7      while pq is not empty do
8          Pop a node n_min from pq, and add it to Visited and ConQueue;
9          Create a new node n along the same trajectory in n_min;
10         if n.t_c ≥ t and not (∃n' ∈ Visited that n' ≻ n) then
11             Add n to pq;
12 return the edges in Visited as RA;
```



(a) Trajectory Database (b) Fastest Trajectory (c) Trajectory Database (d) SG Index (e) Edge Neighbors

Fig. 4. Inspiration for *SG-index*.

4.2 Skip Graph Index

Observation. In essence, TE and TE+ enumerate all possible trajectory connections. However, it is not necessary to keep all trajectories and explore every possible trajectory connection. For example, as shown in Fig. 4(a), we have four trajectories in different colors and time costs. We do not need to explore any trajectory connection with tr_4, as any trajectory connection containing tr_4 can be replaced by tr_3 with a better time cost. Keeping tr_4 here only increases the computation cost. Furthermore, for each pair of origin and destination (OD), we only need to keep track of the fastest sub-trajectory. Figure 4(b) gives all of the fastest trajectories extracted from Fig. 4(a) based on different OD pairs.

Theorem 2 *For any edge e_i in the real-time reachable area, it can be reached from the query location by connecting no more than k sub-trajectories, where each sub-trajectory is the fastest one between its origin and destination.*

Proof. Each qualified edge is reachable from q via at least one qualified ctr, which can be segmented into no more than k sub-trajectories. By connecting the OD of each sub-trajectory with the fastest sub-trajectory between them, we can create a new connected trajectory ctr', where $Cost(ctr') \leq Cost(ctr)$ and $D(ctr') \leq D(ctr)$. If $D(ctr') < D(ctr)$, there at least exists two neighbor sub-trajectories in ctr' belonging to the same trajectory.

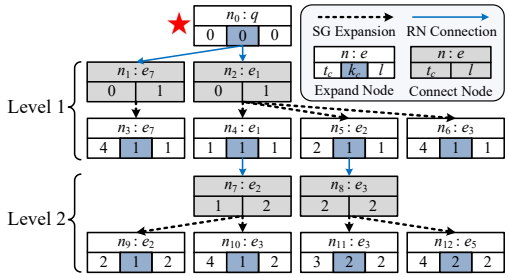


Fig. 5. An Example of *SGE-tree* with $t = 4$ minutes and $k = 2$.

Data Structure. With the insight above, we propose Skip Graph index (*SG-index*), which preserves the fastest sub-trajectories connecting every OD pair. Indeed, *SG-index* is a weighted directed graph, in which a node (*SG-node*) is a road segment on road networks, an edge (*SG-link*) connecting two *SG-nodes* e_i and e_j represents there is at least one sub-trajectory traveling from $e_i.v_{start}$ to $e_j.v_{end}$, and the weight of an *SG-link* is the minimum time cost on it. Figure 4(d) is the *SG-index* of the trajectory database demonstrated in Fig. 4(c).

Construction. *SG-index* is constructed by scanning trajectories. For each trajectory, all of its sub-trajectories are examined to create *SG-links*. The weight of an *SG-link* is assigned as the time cost of the fastest sub-trajectory traversing it. The time complexity of *SG-index* construction is $\mathcal{O}(m \times n^2)$, where m is the number of trajectories in $\mathcal{T}$, and n is the average length of each trajectory. *SG-index* stores the minimum time cost of sub-trajectories in a time period, e.g., the most recent 30 minutes, so it cannot be updated incrementally with new trajectory updates. Instead, it needs to be rebuilt periodically, e.g., every one minute. We can also deploy a distributed streaming framework, like Flink of Storm, to reduce the construction time.

Query Processing. We propose *SGE-tree* (Skip Graph Expansion tree) to find all k-hop neighbours of q based on *SG-index* and road networks, where *SG-index* provides the minimum time cost information between two edges, and road networks give the hints of trajectory connections. Figure 5 is the *SGE-tree* based on Fig. 4(c), which is organized into k levels with the query location as root. *SGE-tree* consists of two types of nodes and two types of links: 1) ***Connect Node.*** This node (marked in grey) is generated based on the neighbour of road segments, with four properties: an identifier n, an edge e, a time cost t_c, and a level number l. 2) ***Expand Node.*** This node (marked in white) is generated based on the expansion of *SG-index*. It contains five properties: an identifier n, an edge e, a time cost t_c, a degree cost k_c, and a level number l. 3) ***Road Network Connection.*** RN connection (blue solid arrow) connects an *expand node* to a *connect node*, based on the neighbours of road segments. Along this type of link, the nodes $(n_i \& n_j)$ have the same time cost $n_j.t_c = n_i.t_c$, but an increasing level number $n_j.l = n_i.l + 1$. 4) ***SG Expansion.*** This link (black dotted arrow) connects a *connect node* to an *expand node*, based on the neighbours of *SG-nodes* in *SG-index*. Along this type of link, the nodes $(n_i \& n_j)$ have an increasing time cost $n_j.t_c = n_i.t_c + Cost(n_i.e \to n_j.e)$, and the same level number $n_j.l = n_i.l$, where $Cost(n_i.e \to n_j.e)$ is the weight from $n_i.e$ to $n_j.e$ in *SG-index*.

Note that the level number l on an *expand node* is not equivalent to its degree cost k_c. l means the i-th hop neighbours of the root, but multiple hops in *SGE-tree* may belong to the same trajectory. For example in Fig. 5, $n_9.l = 2$, as it is a two hop neighbour from q. However, both of the hops are the sub-trajectories of the red dotted trajectory in Fig. 4(c), i.e., $tr_{red}[e_1...e_1] \rightarrow tr_{red}[e_2...e_2]$, which makes the degree cost only one. As a result, we know that in *SGE-tree*, $l \geq k_c$.

Theorem 3 *An SGE-tree with the level number of k covers all qualified edges e_i in reachable area $RA(\mathcal{T}, q, t, k)$.*

Proof. Suppose edge e is in the reachable area RA, but does not appear in *SGE-tree* with a level of k. As e does not appear in *SGE-tree* with a level of k, it means that e cannot be reached from q by connecting any k fastest sub-trajectories. In other words, to reach e, more than k fastest sub-trajectories should be connected. Thus, it disqualifies e to be reachable, which is contradictory to our assumption.

Therefore, finding a reachable area with k trajectory connections is equivalent to finding the k-hop neighbors of q in *SG-index*. A basic idea is to search the *SGE-tree* level by level using a breath-first search, as this order guarantees the trajectory connections with a smaller degree cost is searched first (denoted as **SGE**). However, we can observe that there are still redundant computations. For example in Fig. 5, n_9 should not be searched when n_5 exists, as they have the same time cost and edge. We can avoid this situation based on *node domination*.

Definition 5 (Node Domination in SGE-tree) *Given two expand nodes n_i and n_j, if $n_i.e = n_j.e$, $n_i.t_c \leq n_j.t_c$, and $n_i.l \leq n_j.l$, then n_i dominates n_j, denoted as $n_i \succeq n_j$.*

Lemma 1 *If an expand node n_j in an SGE-tree has $n_j.l > n_j.k_c$, there must exist an expand node n_i in level $n_j.k_c$, with $n_i.k_c = n_i.l = n_j.k_c$ and $n_i.t_c = n_j.t_c$.*

Although l is not equivalent to k_c, we can still use the domination relation to prune the disqualified nodes, when applying the breath-first search.

Theorem 4 *If there exist two expand nodes n_i and n_j such that $n_i \succeq n_j$, then n_j and all its children can be pruned, when using the breath-first search.*

Proof. Suppose $n_i \succeq n_j$, then $n_i.e = n_j.e$, $n_i.t_c \leq n_j.t_c$ and $n_i.l \leq n_j.l$. There are two possible cases between $n_i.k_c$ and $n_j.k_c$: 1) $n_i.k_c \leq n_j.k_c$, in this case, n_j can be pruned, as all the children of n_j can be attached to n_i; or 2) $n_i.k_c > n_j.k_c$, in this case, $n_j.k_c \neq n_j.l$. Otherwise, if $n_j.k_c = n_j.l$, we will have $n_i.l \geq n_i.k_c > n_j.k_c = n_j.l$, which contradicts to the domination relation $n_i.l \leq n_j.l$. Thus, $n_j.k_c < n_j.l$. According to Lemma 1, there must exist a node in level $n_j.k_c$ that covers the same trajectory connection. As a result, we can safely remove n_j and all its children from further expansion.

According to Theorem 4, we propose **SGE+** (Algorithm 2) to prune all disqualified *expand nodes* based on SGE. SGE+ performs a k-iteration process, where each iteration executes two functions: 1) *RNConnection*, which creates *connect nodes* based on the road network neighbours of *expand nodes* in the previous level; and 2) *SGExpansion*, which identifies qualified *expand nodes* in this level based on the links in *SG-index* and the *connect nodes* in the previous

Algorithm 2: SGE+

Input: *SG-index* of $\mathcal{T}$, query location q, time constraint t, degree constraint k.
Output: Reachable area $RA(\mathcal{T}, q, t, k)$.

Init a key-value store $Edge2MinT$ to track the min time cost of edges from q;
Init two sets Exp and Con to store *expand nodes* and *connect nodes* in a level;
Form the root of *SGE-tree* with q, and add it to Exp;
for $i = 1$ *to* k **do**
    `// RNConnect Step`
    Pop all *expand nodes* in Exp, create new *connect nodes* based on the road network neighbours, and add them to Con;
    `// SGExpansion Step`
    while Con *is not empty* **do**
        Pop a node n_c from Con;
        Create a new *expand node* n_e based on n_c and *SG-index*;
        if $n_e.t_c \leq t$ *and* $Edge2MinT[n_e.e] > n_e.t_c$ **then**
            Add n_e to Exp; $Edge2MinT[n_e.e] = n_e.t_c$;
return the edges in $Edge2MinT$ as RA;

step. We discard the disqualified *expand node* if either it has a time cost more than t or its edge has been searched before with a smaller time cost.

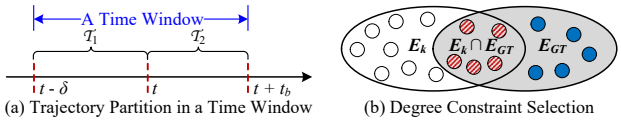
It is worth noting that we only leverage the time costs t_c and the level numbers l of *expand nodes* to perform the pruning process. As a result, in implementation, it is unnecessary to store the degree costs k_c in *expand nodes*.

5 Model Learning & Prediction

The degree constraint k is intangible for users. We cannot assign a fixed k at all places and all times, as k is affected by various external factors. To this end, we propose to dynamically predict the k value in any location and at any time.

5.1 Label Generation

One of the challenges to predict k is that, there is no label of reachable areas in our dataset. As k is a trade-off between coverage and reliability, a bigger k achieves a higher coverage, but results in a lower reliability. The intuition is to get a reasonable coverage with the k as small as possible. As a result, we generate the labels of k using historical trajectories. More specifically, we regard the reachable area without any trajectory connection based on "future" trajectories as partial ground truth, and find reachable areas with different k values using "recent" trajectories. The minimum k that satisfies a coverage threshold is set as the label. To get labels for a time budget t_b using the trajectories in most recent time δ, three tasks are performed: 1) **Trajectory Partition.** The historical trajectories are partitioned by a sliding window of size $\delta + t_b$. The trajectories in a time window are further divided into two sets, $\mathcal{T}_1$ and $\mathcal{T}_2$, as shown in Fig. 6(a). 2) **Reachable Area Discovery.** In each time window, we take each edge e at the time t as a start location. For each $k \in \{1, 2, ..., 5\}$, a reachable area $E_k = RA(\mathcal{T}_1, e, t_b, k)$ with trajectory connections is discovered, using the techniques introduced in Section 4. Besides, we find the reachable area E_{GT} starting from e without any trajectory connection as the partial ground truth



(a) Trajectory Partition in a Time Window (b) Degree Constraint Selection

Fig. 6. Illustration of Label Generation.

of the real reachable area, using the technique in [9]. 3) ***k*-Selection.** As shown in Fig. 6(b), for each $k \in \{1, 2, ..., 5\}$, we calculate the ratio between $|E_k \cap E_{GT}|$ and $|E_{GT}|$, where $|*|$ is the cardinality of a set. We then select the minimum k as the label k_l that makes the ratio greater than η, $0 \leq \eta \leq 1$, formally defined as Equ. (3). It means that E_k covers the most edges in E_{GT}, but k is as small as possible. To achieve a high reliability, we set $\eta = 0.9$ in implementation.

$$k_l = \min k, \quad s.t. \ |E_k \cap E_t| \,/\, |E_t| \geq \eta \text{ and } k \in \{1, 2, ..., 5\} \tag{3}$$

5.2 Feature Extraction

We identify five types of features from multiple data sources: 1) **Traffic Features.** For each road segment, we extract two traffic features, i.e., traffic flow and average speed, from the nearby real-time trajectories. 2) **Time Features.** The time of day, day of the week, and holidays are extracted, to capture the periodicity of traffic conditions. 3) **Meteorological Features.** We extract the meteorological features of each query location, such as rainfall, temperature and weather conditions (e.g., cloudy, sunny and rainy). 4) **POI Features.** We calculate the POI distribution within 1km of each query location. The POIs are categorized into food, shopping, company and etc. 5) **Road Network Features.** The structure of road networks affects the traffic conditions. For each road segment, we extract the features from nearby road networks, including intersection number and the length of each road level (e.g., highway, main road, side road and so on).

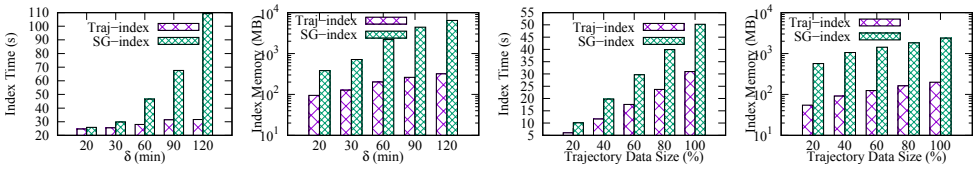
5.3 Model Training

The extracted features are first standardized, and then fed into the-state-of-art model ST-ResNet [10], as it can capture the spatial dependencies, temporal dependencies, and external factors of the traffic conditions. Although k is discrete, we regard this problem as a regression instead of a classification, because the penalties should be different for different predicted k values. For example, if the label is 2, it is better to predict k as 3 than 5. For each discrete $t \in [1, 20]$, we train a model individually. The model that is closest to the given continuous time budget is used when predicting.

6 Evaluation

6.1 Datasets & Settings

Datasets. We adopt four real datasets in our experiments: 1) **Road Networks.** The road networks of Shanghai, China are extracted from OpenStreetMap with 333,766 vertices and 440,922 road segments. 2) **POIs.** We extract the POIs of Shanghai from OpenStreetMap, which contains 1,111,188 records. 3) **Meteorology.** We collect the meteorological data in Shanghai ranging from Dec. 23rd to Dec. 30th, 2016. The data is updated every hour. 4) **Trajectories.** We extract



(a) Time wrt. δ. (b) Memory wrt. δ. (c) Time wrt. Data %. (d) Memory wrt. Data %.

Fig. 7. Indexing Performance.

the taxi trajectories from Dec. 23rd to Dec. 30th, 2016 in Shanghai. It contains 303,673,097 GPS points of 5,669 taxis, whose average sampling rate is 10 seconds. The trajectories generated in most recent δ minutes to the query time is used to simulate the real-time trajectory updates.

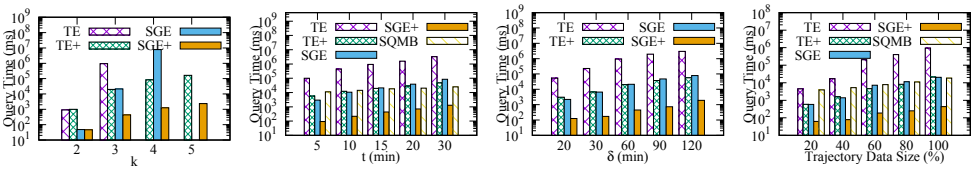
Comparing Methods. We compare our proposed method (i.e. SGE+) with its variants (i.e., TE, TE+ and SGE) and two advanced methods: 1) **SQMB** [9], which finds reachable areas using historical trajectories; and 2) **TTE**, which first estimates the travel time of each road segment [4], then discovers reachable areas based on network expansion method [2]. We also verify the effectiveness of ST-ResNet for our problem, comparing with multiple models including GBDT, RF, SVR and XGBoost.

Experimental Settings. We focus on the efficiency of indexing and query processing (implemented in C#), and the effectiveness of k value prediction (implemented in Python). We randomly select 100 edges as query locations and calculate the average query processing time. 70% of trajectory and meteorology data are used for k value model training, and the left are used for validation. All experiments are performed on a 64-bit Windows Server 2012 with octa-core 2.2GHz CPU and 56 GB RAM. If not specified, we set the default real-time window $\delta = 60$ minutes, time budget $t = 15$ minutes, and degree constraint $k = 3$. Besides, we use 100% available real-time taxi trajectory data by default.

6.2 Indexing Performance

Different real-time windows. Figure 7(a) depicts the indexing time of *Traj-index* and *SG-index* with different real-time windows δ. There are two observations: 1) with a bigger δ, both *Traj-index* and *SG-index* need more time to build, as we need to process more trajectories; 2) compared with *Traj-index*, the indexing time of *SG-index* increases more significantly with a larger δ, as more sub-trajectories are examined to update *SG-index*. Figure 7(b) shows the memory usage of *Traj-index* and *SG-index* with the increasing real-time window δ. It is clear that more spaces are used for both indexes. Moreover, the space consumed by *SG-index* grows exponentially with a larger δ, as longer trajectories are generated, which creates exponentially more sub-trajectory candidates to create the links in *SG-index*.

Different trajectory data sizes. Figure 7(c) presents the construction time for two indexes, where the dataset contains different numbers of trajectories randomly sampled from 20% to 100%. It is observed that the indexing time of both indexes grows linearly with an increasing sample ratio, because for both indexes, they need to scan the dataset for one time. Moreover, *SG-index* consumes



(a) Time wrt. k. (b) Time wrt. t. (c) Time wrt. δ. (d) Time wrt. Data %.

Fig. 8. Query Processing Performance.

much more time, as it needs to check all sub-trajectories to create the links. Figure 7(d) indicates that the memory usage of both indexes increases with more trajectories. It is interesting to see that the memory growth of *SG-index* is slower comparing to different δ, because more trajectories introduce a limited number of sub-trajectories with distinct OD pairs as the links in *SG-index*.

We do not compare the indexing performance with SQMB and TTE here, as SQMB scans all historical trajectories when building indexes (which is time-consuming), and TTE does not build indexes. Besides, in the next subsection, we do not compare the query efficiency of TTE, because it is unfair for TTE if we consider its prediction time, which is costly.

6.3 Query Processing Performance

Different degree constraints. Figure 8(a) shows the query processing time with different k, from 2 to 5 ($k = 1$ is not tested, as it does not involve any trajectory connection). With an increasing k, the query processing time of all methods increases. Moreover, TE+ (or SGE+) is more efficient than TE (or SGE), as redundant computations are avoided. Furthermore, SGE takes more time than TE+ when k is large. Because with more combinations of sub-trajectories, pruning the disqualified nodes in *TE-tree* or *SGE-tree* is more effective. In fact, TE is not able to compute the results when $k \geq 4$. Similarly, SGE also fails when $k \geq 5$. SQMB is not tested here as it does not involve trajectory connections.

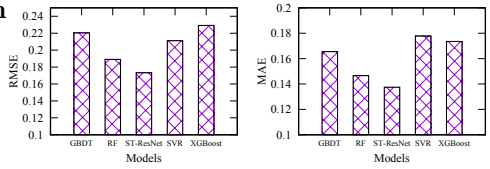
Different time constraints. As depicted in Fig. 8(b), with the growth of t, the query processing time of all methods increases. It is clear that with a larger t, more candidate road segments are tested. We can also notice that TE+ (or SGE+) is much better than TE (or SGE), and SGE+ is the most efficient. It is interesting to see that the performance of TE+ exceeds SGE when t is large, as each pruned candidate leads to a longer (i.e., with more t) redundant search process. SQMB is faster than TE, TE+ and SGE when t is larger, which proves the big challenges with trajectory connections. However, thanks to the effective indexing and pruning techniques, SGE+ is much faster than SQMB in all cases.

Different real-time windows. Figure 8(c) indicates that with a larger time window δ, all methods take more time, as more road segments are included in a trajectory, leading to a larger *TE-tree* or *SGE-tree*. Here we do not test SQMB as it uses all historical trajectories, which is not affected by the real-time window.

Different trajectory data sizes. Figure 8(d) shows that the query processing time increases with more trajectories, as more trajectory connection candidates are tested. TE+ is comparable to SGE, because the pruning techniques based on node domination play a major factor in improving the querying efficiency.

6.4 Effectiveness of k Prediction

Figure 9 shows the average RMSE (Root Mean Square Error) and MAE (Mean Absolute Error) of different models, which indicates that ST-ResNet is the best model for our problem, in terms of both RMSE and MAE. Because ST-ResNet not only captures the temporal closeness, period, and trend properties of traffic conditions, but also model the spatial dependency among different locations.



(a) RMSE wrt. Models. (b) MAE wrt. Models.

Fig. 9. Effectiveness of k Prediction.

6.5 Case Study

(a) 2016/12/23 19:00 (SGE+) (b) 2016/12/30 19:00 (SGE+) (c) 2016/12/30 19:00 (SQMB) (d) 2016/12/30 19:00 (TTE)

Fig. 10. A Case of Concert ($t = 5$ minutes, $\delta = 30$ minutes).

Figure 10 shows the reachable areas in the Mercedes-Benz Arena, Shanghai at the same time on two different days using different methods. Although both days are Friday, the reachable area in Fig. 10(b) is much smaller than that in Fig. 10(a), because there is a concert in the arena at 19:30, Dec. 30th, 2016[6]. More than 10,000 fans gathered here, causing a heavy traffic jam. As a result, our solutions reflect the traffic jam, where the reachable area only covers the nearby road segments. Comparing to SGE+, SQMB gives the same reachable area in all days as shown in Fig. 10(c), and TTE gives a reachable area as shown in Fig. 10(d), thus they can hardly capture the real-time traffic conditions such as events. Besides, SQMB could miss some reachable road segments if there is no trajectory that exactly traverses from the query location to them (i.e., the orange area). However, the trajectory connection techniques proposed by this paper can mitigate this situation.

7 Related Works

Reachability Query. The conventional reachability query is one of the fundamental graph operations, asking if two nodes are connected in a directed graph [17–23]. These works can be categorized into two main categories: 1) reachability query on static graphs, e.g., [17] introduces a graph reduction method, while other works [23, 20] propose different labeling methods to reduce the index size; and 2) reachability query on dynamic graphs, whose edges and vertexes change over time. For example, [22] proposes different indexes to efficiently handle vertex insertions and deletions. The conventional reachability query problem is very different from our real-time reachable area discovery task, as their reachability only considers the graph structure. The closest work is [9], which finds

[6] http://bit.ly/2y6f3BF

reachable areas based on massive historical trajectories that passed the query location during the request hour. By analyzing the daily statistics of the qualified trajectories, the reachable area with a certain probability can be identified. However, this method cannot capture weather, traffic conditions and events, which is not suitable for real-time reachable area discovery.

Travel Time Estimation. Travel time estimation calculates the time cost on a given path. [4–7] leverage the readings of loop detectors or trajectories to infer the time cost on each road segment. Then, the time cost of a path is estimated by summing up all costs of the road segments along the given path. These works ignore the dependencies between road segments. To capture the delays of road intersections/traffic lights and improve the estimation accuracy, [24–26] estimate the travel time of a path by considering the trajectories passed the entire path, and [27] proposes an end-to-end deep learning framework to estimate the travel time. The techniques of travel time estimation cannot be applied directly to the discovery of real-time reachable areas, as they require the predefinition of a path, including the origin and destination locations. In the scenario of reachable area discovery, the destinations and the paths from the query location are not predefined. As a consequence, directly applying travel time estimation methods requires to examine all possible destinations and possible paths to them, which is inefficient and infeasible in a real-time scenario.

8 Conclusion

This paper provides the first attempt to discover real-time reachable areas with dynamic trajectory connections. A framework that combines indexing techniques with machine learning is proposed. Our proposed indexing and query processing methods can efficiently find real-time reachable areas with an arbitrary number of trajectory connections. We also propose to predict the best connection number that achieves a good coverage while guarantees reliability. Extensive experiments and one case study on four real datasets confirm the effectiveness and efficiency of our proposed methods for the real-time scenarios.

Acknowledgement. This work was supported by NSFC(No. 61976168, No. 61672399, No. U1609217) and the Science Foundation of Hubei Province (No. 2019CFA025). We would like to thank Yuxuan Liang to discuss the label generation methods of connection number.

References

1. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: an efficient and robust access method for points and rectangles," in *SIGMOD*, vol. 19, no. 2. ACM, 1990, pp. 322–331.
2. D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," in *VLDB*, 2003, pp. 802–813.
3. V. Bauer, J. Gamper, R. Loperfido, S. Profanter, S. Putzer, and I. Timko, "Computing isochrones in multi-modal, schedule-based transport networks," in *ACM SIGSPATIAL*, 2008, pp. 1–2.
4. J. Wang, Q. Gu, J. Wu, G. Liu, and Z. Xiong, "Traffic speed prediction and congestion source exploration: A deep learning method," in *ICDM*. IEEE, 2016, pp. 499–508.

5. D. Wang, W. Cao, M. Xu, and J. Li, "Etcps: An effective and scalable traffic condition prediction system," in *DASFAA*. Springer, 2016, pp. 419–436.
6. X. Lin, Y. Wang, X. Xiao, Z. Li, and S. S. Bhowmick, "Path travel time estimation using attribute-related hybrid trajectories network," in *CIKM*. ACM, 2019, pp. 1973–1982.
7. Q. Xie, T. Guo, Y. Chen, Y. Xiao, X. Wang, and B. Y. Zhao, ""how do urban incidents affect traffic speed?" a deep graph convolutional network for incident-driven traffic speed prediction," *arXiv preprint arXiv:1912.01242*, 2019.
8. S. Skiena, "Dijkstra's algorithm," *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pp. 225–227, 1990.
9. G. Wu, Y. Ding, Y. Li, J. Bao, and Y. Zheng, "Mining spatio-temporal reachable regions over massive trajectory data," in *ICDE*. IEEE, 2017, pp. 1283–1294.
10. J. Zhang, Y. Zheng, D. Qi, R. Li, X. Yi, and T. Li, "Predicting citywide crowd flows using deep spatio-temporal residual networks," *Artificial Intelligence*, vol. 259, pp. 147–166, 2018.
11. S. Ruan, R. Li, J. Bao, T. He, and Y. Zheng, "Cloudtp: A cloud-based flexible trajectory preprocessing framework," in *ICDE*. IEEE, 2018.
12. J. Bao, R. Li, X. Yi, and Y. Zheng, "Managing massive trajectories on the cloud," in *SIGSPATIAL*. ACM, 2016, p. 41.
13. R. Li, S. Ruan, J. Bao, and Y. Zheng, "A cloud-based trajectory data management system," in *ACM SIGSPATIAL*, 2017, pp. 1–4.
14. R. Li, S. Ruan, J. Bao, Y. Li, Y. Wu, L. Hong, and Y. Zheng, "Efficient path query processing over massive trajectories on the cloud," *IEEE Transactions on Big Data*, 2018.
15. R. Li, H. He, R. Wang, Y. Huang, J. Liu, S. Ruan, T. He, J. Bao, and Y. Zheng, "Just: Jd urban spatio-temporal data engine," in *ICDE*. IEEE, 2020.
16. R. Li, H. He, R. Wang, S. Ruan, Y. Sui, J. Bao, and Y. Zheng, "Trajmesa: A distributed nosql storage engine for big trajectory data," in *ICDE*. IEEE, 2020.
17. J. Zhou, S. Zhou, J. X. Yu, H. Wei, Z. Chen, and X. Tang, "Dag reduction: Fast answering reachability queries," in *SIGMOD*. ACM, 2017, pp. 375–390.
18. L. D. Valstar, G. H. Fletcher, and Y. Yoshida, "Landmark indexing for evaluation of label-constrained reachability queries," in *SIGMOD*. ACM, 2017, pp. 345–358.
19. S. Anirban, J. Wang, and M. S. Islam, "Multi-level graph compression for fast reachability detection," in *DASFAA*. Springer, 2019, pp. 229–246.
20. J. Su, Q. Zhu, H. Wei, and J. X. Yu, "Reachability querying: Can it be even faster?" *TKDE*, vol. 29, no. 3, pp. 683–697, 2017.
21. M. Sarwat and Y. Sun, "Answering location-aware graph reachability queries on geosocial data," in *ICDE*. IEEE, 2017, pp. 207–210.
22. H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *ICDE*. IEEE, 2016, pp. 145–156.
23. H. Wei, J. X. Yu, C. Lu, and R. Jin, "Reachability querying: an independent permutation labeling approach," *VLDBJ*, vol. 27, no. 1, pp. 1–26, 2018.
24. Y. Wang, Y. Zheng, and Y. Xue, "Travel time estimation of a path using sparse trajectories," in *SIGKDD*. ACM, 2014, pp. 25–34.
25. J. Dai, B. Yang, C. Guo, C. S. Jensen, and J. Hu, "Path cost distribution estimation using trajectory data," *VLDB*, vol. 10, no. 3, pp. 85–96, 2016.
26. J. Xu, Y. Zhang, L. Chao, and C. Xing, "Stdr: A deep learning method for travel time estimation," in *DASFAA*. Springer, 2019, pp. 156–172.
27. D. Wang, J. Zhang, W. Cao, J. Li, and Z. Yu, "When will you arrive? estimating travel time based on deep neural networks," in *AAAI*, 2018.