Data Science and Engineering (2023) 8:1–22
https://doi.org/10.1007/s41019-023-00205-y

RESEARCH PAPER

Construct Trip Graphs by Using Taxi Trajectory Data

Hao Yu[1,2] · Xi Guo[1,2] · Xiao Luo[3,4] · Weihao Bian[3,4] · Taohong Zhang[1,2]

Received: 1 November 2022 / Revised: 13 January 2023 / Accepted: 4 February 2023 / Published online: 18 February 2023

Abstract
The trip graph, which can model the residents' taxi demands, consists of vertices used to indicate trips and edges used to indicate the follow-up relationships between trips. A trip v_j is a tight follow-up trip of another trip v_i, if a taxi can arrive at the departure location of v_j within a time threshold δ after it finishes v_i. However, for big cities, there are a large amount of trips every day and it is time consuming to construct a trip graph. In this paper, we propose efficient algorithms to construct trip graphs for big cities. When constructing a trip graph, the most expensive step is to connect the vertices if the tight follow-up relationships exist. To find out the tight follow-up trips fast, we design an index considering both spatial and temporal constraints. To designate appropriate search areas, we propose efficient methods to determine the distance-based search areas and the traffic-based search areas. We conduct experiments on real datasets, i.e., the taxi trajectories of Shanghai in 2015. The experimental results show that our algorithm can construct the trip graph about 40 times faster than the straightforward method. We also demonstrate the usages of the trip graph in green transportation applications, i.e., the minimum fleet analysis and the minimum total "idle" mileage analysis.

Keywords Trip graph · Trip index · Spatial queries · Trajectory data

1 Introduction

Governments and taxi companies collect a huge number of trajectory data by using positioning devices (for example, GPS, etc.). Travel demands can be mined from the trajectory data. A trip graph [1] is used to express the travel demands in a concise way. In the graph, each vertex denotes a trip, and the directed edge from vertex v_i to vertex v_j denotes that v_j could be a follow-up trip of v_i. For example, in Fig. 1a, there are nine trips $\{v_1, v_2, ..., v_9\}$. Each trip has its departure location and time and its arrival location and time. Figure 1(b) shows the trip graph of the nine trips. In the graph, each vertex denotes a trip and the edge connect two vertices denotes the follow-up relationship between two trips. The outgoing edge from v_1 indicates a driver has enough time to reach the departure location of v_2 after he finishes v_1. In the case of a taxi-hailing app, the trip graph maintains the sequence of passengers' orders. For example, after a taxi driver finishes order v_2, she can take one order among v_3, v_4 and v_5 as the next order. If she chooses v_4 and finishes it, she can continue to take v_9.

A trip graph is the basis of taxi fleet managements [2] and car exhaust analyses [3]. Researchers can use one day's trip graph to compute the minimum number of taxis that can meet daily demands of residents in a city. In fact, the problem of minimizing the number can be converted to finding the minimum path cover on the trip graph. Using CopertIII model [4], which is a model to estimate the volumes of CO_2 emissions of vehicles, researchers can further

✉ Xi Guo
xiguo@ustb.edu.cn

Hao Yu
348009205@qq.com

Xiao Luo
luo.xiao@tongji.edu.cn

Weihao Bian
bianwh@tongji.edu.cn

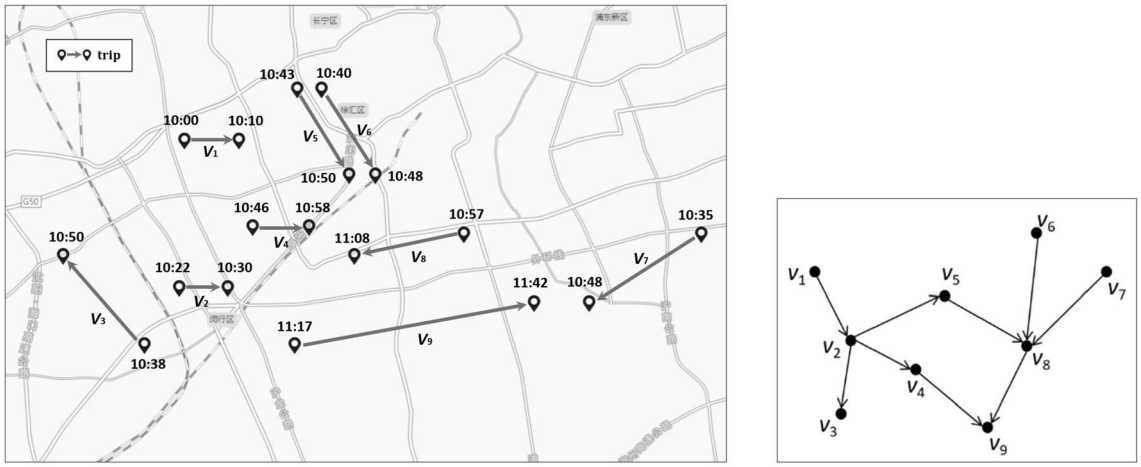
Taohong Zhang
taohzhang@ustb.edu.cn

1 School of Computer and Communication Engineering, University of Science and Technology Beijing, NO. 30 Xueyuan Road, Beijing 100083, China

2 Beijing Key Laboratory of Knowledge Engineering for Materials, NO. 30 Xueyuan Road, Beijing 100083, China

3 College of Transportation Engineering, Tongji University, NO. 4800 Caoan Road, Shanghai 201804, China

4 Urban Mobility Institute, Tongji University, NO. 1239 Siping Road, Shanghai 200092, China



(a) Trips (b) A trip graph

Fig. 1 An example of trips and its trip graph

compute the CO_2 reduction if the taxi fleet size decreases to the minimum number. However, it is time consuming to construct a trip graph for a large number of trips. In this paper, we study how to construct a trip graph efficiently.

The number of taxi trips in a large city (for example, shanghai, China) is about 300,000 one day. A trip graph models the citizens' travel demands in a whole day and it consists of a large number of vertices (about 300,000) which indicate the trips. A trip (i.e., vertex) should be connected with its **tight follow-up trips** with directed edges. Given two trips v_i and v_j, if the time interval between v_i and v_j is not too long and a driver can travel from v_i's arrival location to v_j's departure location in time, v_j is a tight follow-up trip of v_i. In this paper, our problem is to find out all tight follow-up trips of each trip. A straightforward way is checking trips one by one and identifying the tight follow-up trips of each trip. The time complexity of this method is $O(n^2)$, where n is the number of trips and n may be very large (for example, 300,000).

To find out tight follow-up trips more efficiently, we propose a trip index by considering the spatial and temporal constraints. The spatial constraint is that the distance between two trips cannot be too far away. The temporal constraint is that a driver needs enough time to travel to the destination. The trip index consists of an array of time slots and several R^*-trees. The array of time slots is a one-dimensional array, which records information such as the time boundary of each time slot. R^*-trees record the locations of all trips in a single time slot. The array of time slots determines the content of R^*-trees.

We can find out the tight follow-up trips of one trip v_i efficiently by using the trip index. First, we identify candidate time slots based on the arrival time of v_i. Then we perform a spatial area search on each candidate time slot to find tight follow-up trips. In the search procedure, to designate an appropriate search area is very important. We study the distances-based search area and the traffics-based search area. The distance-based search area depends on the travel time and speed of the vehicle. If we allow drivers to drive longer, vehicles can reach a greater range. However, the traffic conditions of different road sections are different, and the speed of vehicles is determined in the traffic grid. The traffic-based search area depends on the traffic grid. We map trajectories into cells and calculate the average speed. We implement a breadth-first search on the traffic grid, and find the traffic-based search area. Finally, we aggregate the search area by merging cells. If we designate the search area by considering the distances, we can obtain results fast. However, some results may be not the tight follow-up trips in reality. If we designate the search area by considering the traffics, we can obtain results that are closer to the reality. However the method requires much more time.

We evaluate the perfomances of the proposed index and search algorithms by using real datasets, which is the taxi trajectory data in Shanghai, China, in 2015. The experimental results show that using the trip index we can find out tight follow-up trips about 40 times faster than using the baseline methods. The experimental results also show that we can reduce false tight follow-up trips by nearly 40% if we designate traffic-based search areas. To illustrate the usage of the trip graph, we demonstrate the experimental results of minimum taxi fleet analyses. Using the trip graph, the minimum path coverage algorithm can estimate the minimum number of taxis that can meet urban residents' daily travel demands [2]. The experimental results show that we only need 12,000 taxis to meet urban residents' daily travel demands of Shanghai. We also calculate the exhaust emissions that can be reduced according to the COPERT

III model [4]. The experimental results show that when the average vehicle speed is 36 km/h, the scheduling method of the minimum taxi fleet can reduce the exhaust emissions by about 15%.

In summary, we make the following contributions:

- We propose a trip index that can organize taxi trips considering their spatial and temporal features.
- To construct a trip graph efficiently, we present a distance-based search algorithm and a traffic-based search algorithm that can find out the tight follow-up trips fast.
- Experiments results on real trajectory datasets show that our algorithms outperforms the straightforward algorithm of constructing trip graphs.

Section 2 defines the problem of constructing trip graphs and introduces the baseline algorithms in details. Section 3 introduces the trip index in details. Section 4 introduces how to find out tight follow-up trips according to distance-based search areas and according to traffic-based search areas, respectively. Section 5 analyzes the experimental results and demonstrates the usages of trip graphs. Section 6 presents a review of related work. Section 7 concludes the paper.

2 Preliminaries

The trip graph is used to model taxi demands of residents. In this section, we define the trip graph and related concepts. At the end of the section, we point out the problem we want to solve.

Definition 1 (Trip graph) A trip graph $G(V, E)$ consists of a vertex set V and an edge set E, where a vertex $v_i \in V$ denotes a trip and an edge from v_i to v_j means v_j is a tight follow-up trip of v_i.

In the definition, a trip v_i is used to model a taxi demand. It consists of four elements, i.e., $(v_i^{start}, v_i^{on}, v_i^{end}, v_i^{off})$, where v_i^{start} and v_i^{on} are the departure time and location of v_i, v_i^{end} and v_i^{off} are the arrival time and location of v_i. The edges of a trip graph indicate the tight follow-up relationships between trips. Next, we define the **follow-up** relationship first and then define the **tight follow-up** relationship.

If a taxi has enough time to reach v_j^{on} (i.e., the departure location of v_j) after she reaches v_i^{off} (i.e., the arrival location of v_i), v_j is a **follow-up trip** of v_i. We give the formal definition as follows.

Definition 2 (Follow-up trip) Assuming a taxi spends at least t_{ij} time in traveling from v_i^{off} to v_j^{on}, if

$$t_{ij} \leq v_j^{start} - v_i^{end}, \tag{1}$$

v_j is a **follow-up trip** of v_i.

In this paper, we estimate t_{ij} by considering the Euclidean distance between v_i^{off} and v_j^{on}, or considering the traffics between the two locations. If v_j is a follow-up trip of v_i, a taxi driver can pick up v_j after she finishes v_i. But in real applications, a taxi driver may be not willing to pick up v_j, if the time interval between the two trips (i.e., $v_j^{start} - v_i^{end}$) is too long. To constrain the time interval, the **tight follow-up** relationship is defined as follows.

Definition 3 (Tight follow-up trip) Given a time interval threshold δ, if v_j is the follow-up trip of v_i and their time interval is not larger than δ, i.e.,

$$t_{ij} \leq v_j^{start} - v_i^{end} \leq \delta \tag{2}$$

v_j is a **tight follow-up trip** of v_i.

In this paper, we set δ to 15 minutes as [2] points out, and we use V_i^* to denote all the tight follow-up trips of v_i.

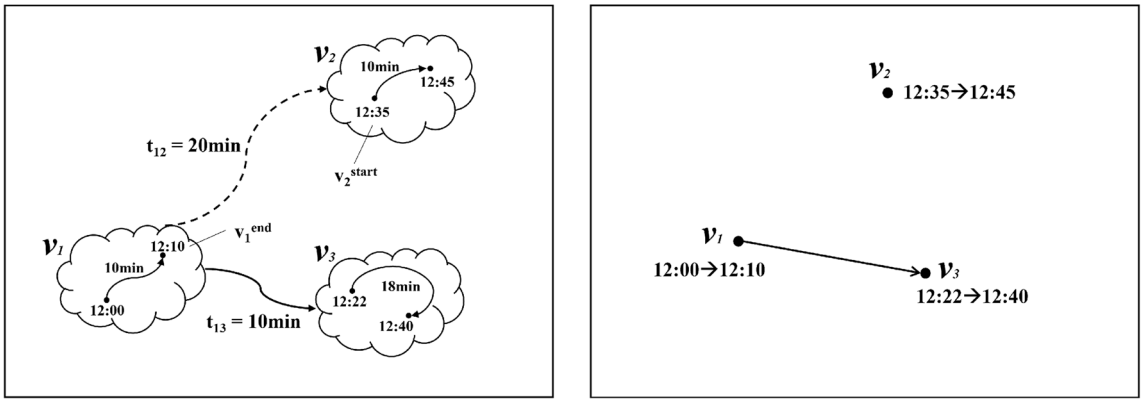
Definition 4 (Trip graph construction problem) Given a trip set V and a time threshold δ, the **trip graph construction problem** is to let trips V be vertices and to connect each trip v_i with its tight follow-up trips V_i^*.

To construct a trip graph, the most time consuming step is to find out the tight follow-up relationships and connect vertices. In this paper, we focus on solving the problem of finding out all the tight follow-up trips V_i^* of each trip v_i efficiently.

Figure 2 shows a toy example. In Fig. 2a, there are three trips v_1, v_2 and v_3. A driver spends 20 minutes in travelling from v_1^{off} to v_2^{on} and she spends 10 minutes in travelling from v_1^{off} to v_3^{on}. Considering the departure time of v_2 and the departure time of v_3, a driver can reach v_2^{on} or v_3^{on} in time after she finishes v_1. Therefore, both v_2 and v_3 are the **follow-up trips** of v_1. Since we set δ to be 15 minutes, only v_3 is a **tight follow-up trip** of v_1. Because the time interval between v_1 and v_2 is larger than δ, i.e., $v_2^{start} - v_1^{end} = 25 minutes$. Figure 2b shows the corresponding *trip graph*. The graph consists of three vertices $\{v_1, v_2, v_3\}$ that indicate trips and an edge from v_1 to v_3 that indicates the tight follow-up relationship .

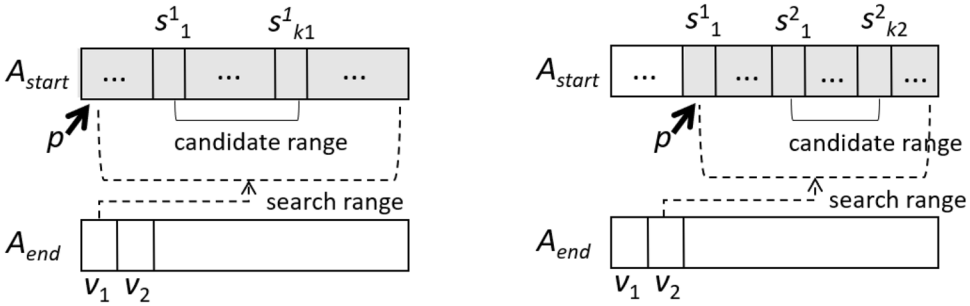
2.1 Baseline Method

To find the tight follow-up trips V_i^* of a trip v_i, a straightforward method is to check $v_j \in V - \{v_i\}$ one by one and determine whether it is a tight follow-up trip according to



(a) Trips (b) A trip graph

Fig. 2 An example of a trip graph



(a) The search scope of v_1 (b) The search scope of v_2

Fig. 3 Sort and Search Algorithm

Definition 3. The time complexity of this method is O(n^2), where n is the total number of trips. In real applications, there are a lot of trips and n is very large. For example, there are about 300,000 taxi trips per day in shanghai.[1] Thus, it is quite time-consuming to construct a trip graph. However, in fact one trip only has a small number of tight follow-up trips. For example, the average number of tight follow-up trips is about 1000 when δ is 15 minutes.[2]

Considering spatial and temporal constraints, we can filter out a large number of trips and have a small number of trips left as candidates. To improve the straightforward method, we propose the **sort and search algorithm**. Firstly, we sort trips in the ascending order of their departure time instants and store their IDs in array A_{start}, and we also sort trips in the ascending order of their arrival time instants and store their IDs in another array A_{end}. Secondly, we look for the candidate trips of each v_i. The candidate trips are the ones whose departure time are between v_i^{end} and $v_i^{end} + \delta$. Since A_{start} is a sorted array, we can find v_i^{end} and $v_i^{end} + \delta$ quickly by using a binary search. Thirdly, we identify the real V_i^* of v_i among the candidates according to Definition 3. This algorithm includes sorting, searching, and verifying. Since the time complexity of sorting is O($n \log_2 n$), the time complexity of searching is O($n \log_2 n$), and the time complexity of verifying is O(nk), where k is the average number of candidates, the time complexity of the whole algorithm is $O(n \log_2 n)$.

To further narrow down the range of each binary search, we use a cursor p to maintain the left position of the next binary search. As Fig. 3a shows, when looking for the candidate trips for v_1, which is the first trip in A_{end}, the cursor p is at the starting element of A_{start}. We have to search for v_1^{end} and $v_1^{end} + \delta$ in the whole A_{start}. Assuming that $\{s_1^1, s_2^1, ..., s_{k_1}^1\}$ are the candidates found, we move cursor p to s_1^1. Thus, in the next iteration, when looking for the candidate trips for v_2, we can narrow down the search range from the whole A_{start} to a part of A_{start}, i.e., from the element pointed by p to the last element in A_{start}, as Fig. 3b shows. Because the departure time of any trip in front of s_1^1 is guaranteed to be earlier than v_2^{end}. In this way, we gradually shrink the binary search range.

[1] See Fig. 13a in Sect 5.1.

[2] See Fig.13b in Sect. 5.1.

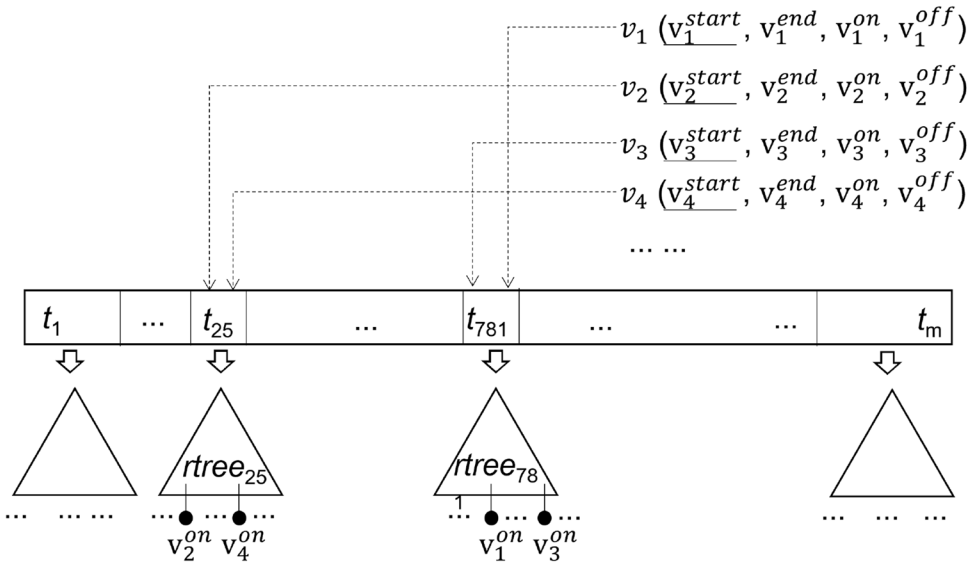


Fig. 4 An example of time slot R^*-tree forest

Algorithm 1: Sort and Search Algorithm

Input: V: the trip set $\{v_1, v_2, \ldots, v_n\}$
Output: $G(V, E)$: the trip graph

```
1  Intialize G(V, E);
2  A_start ← the trip ID array{x_1, x_2, ..., x_i, x_{i+1}, ..., x_n} where
     v_{x_i}^{start} ≤ v_{x_{i+1}}^{start};
3  A_end ← the trip ID array{y_1, y_2, ..., y_j, y_{j+1}, ..., y_n} where v_{y_j}^{end} ≤ v_{y_{j+1}}^{end};
4  p ← 0;
5  for y_j in A_end do
6      id ← y_j;
7      lb ← BinarySearch(v_id^{end}, A_start[p : n]);
8      rb ← BinarySearch(v_id^{end} + δ, A_start[p : n]);
9      p ← lb;
10     for k ← lb to rb do
11         candid ← A_start[k];
12         if v_candid is a tight follow-up trip of v_id then
13             add an edge (v_id, v_candid) to E;
14 return G(V, E);
```

Algorithm 1 summarizes the sort and search algorithm. Line 1 and Line 2 sort the trips according to their departure time and arrival time, respectively. The two arrays A_{start} and A_{end} are formed where elements are trip IDs. Line 4 sets cursor p to be zero which is the subscript of the first element in A_{start}. The outer loop (Line 5 to Line 13) searches for the tight follow-up trips of each trip v_{y_i} according to ID order in A_{end}. Line 7 and Line 8 use binary searches to find out *lb* and *rb* which are the starting subscript and ending subscript of candidate elements in A_{start}. Note that the search range is from p to n. Line 9 updates p in order to shrink the search range in every iteration. Line 12 verifies whether a candidate trip v_{candid} is a tight follow-up trip of v_{id}. If so, Line 13 adds an edge (v_{id}, v_{candid}) to the edge set E. The time complexity of Algorithm 1 is $O(nlog_2m)$ where m is the average length of search ranges, which is about $0.48n$.

3 Constructing Trip Index

Algorithm 1 uses temporal constraint to improve the performance of the algorithm. But it does not consider the spatial constraint. In this section, we build a trip index considering

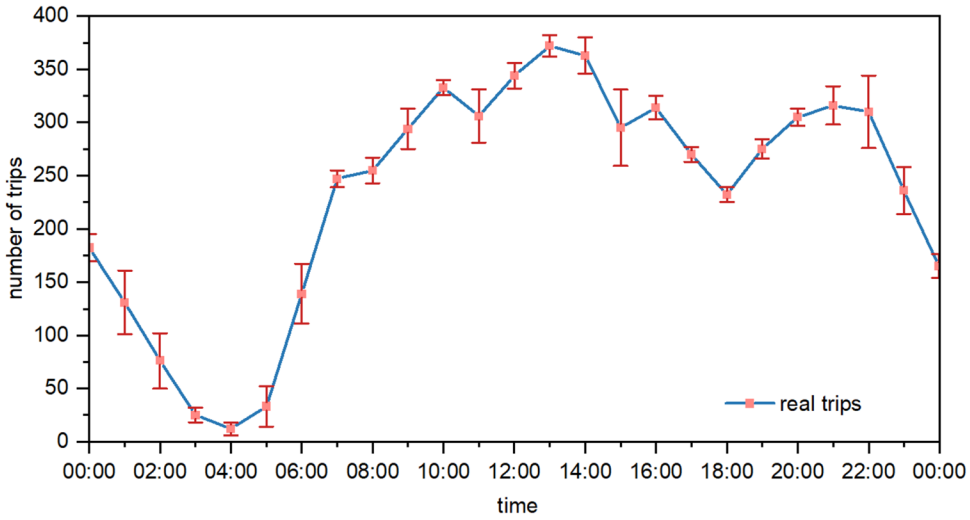


Fig. 5 Number of trips at different time in one day

both spatial and temporal constraints, and in next section we will introduce how to find results fast by using this index.

We divide a day into time slots $\{t_1, t_2, ..., t_m\}$. Each trip is placed into the corresponding time slot t_k according to its departure time v_i^{start}. We set up an R^*-tree index $rtree_k$ for each time slot t_k in order to organize all trips in t_k according to their departure locations (i.e., v_i^{on}'s). As Fig. 4 shows, the whole day is divided into m time slots. Trips $\{v_1, v_3\}$ are assigned to t_{781} and $\{v_2, v_4\}$ are assigned to t_{25} according to their departure time. For each time slot, we build an R^*-tree considering the departure locations. For example, $rtree_{25}$ organizes the departure locations of the trips in t_{25}, and $rtree_{781}$ organizes the departure locations of the trips in t_{781}.

To create time slots, a simple way is to divide a day (24 hours) evenly. For example, the whole day can be divided into 1440 time slots and every time slot has the same length (i.e., 1 minute). However, dividing a day evenly may cause the performances of the index degradates. As Fig. 5 shows, the number of trips fluctuates greatly throughout a day. At the morning peak and the evening peak, the number is much larger than that at the midnight. If we divide the whole day evenly, some time slots may have many trips, while some time slots may have very few trips. The imbalance will cause the height of R^*-tree to be dissimilar. Therefore, we distribute the trips (N in total) evenly into time slots and each time slot contains n_{unit} trips. And the number of time slots is $m = N/n_{unit}$. Since the lengths of time slots are unequal, we use an auxiliary array to record the start time and end time of each time slot.

Algorithm 2: Build a Trip Index

Input: V: the trip set; n_{unit}: the number of trips in each time slot.
Output: $Trees$: the R^*-tree set

```
m ← ⌈V.length/n_unit⌉;
Trees ← ∅;
// Trees consists of m R*-trees, i.e., Trees = {tree_1, ..., tree_m}.
for i ← 1 to m do
    Initialize tree_i;
    Trees ← Trees ∪ tree_i;
V_sorted ← Sort(V);
// Sort by departure time of trip v_i.
for v_i ∈ V_sorted do
    k ← ⌈i/n_unit⌉;
    Insert v_i's departure location v_i^on to tree_k;
```

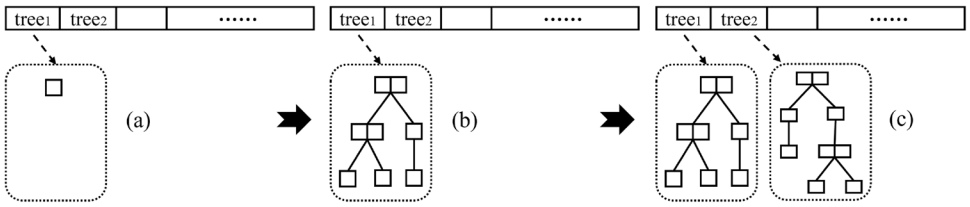


Fig. 6 The process of constructing a trip index

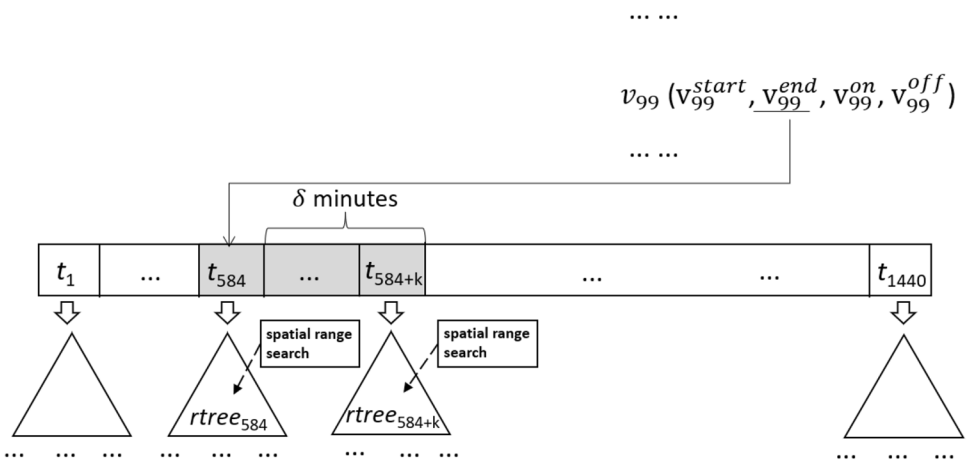


Fig. 7 Search for trips considering both time and spatial constraints

Algorithm 2 describes the procedure of building the trip index. Line 1 calculates the number of time slots divided. Line 2 initializes the R^*-tree set. Line 3 to line 5 use a loop to initialize each *tree* of *Trees*. Line 6 uses function Sort() to sort V by departure time of v_i. Line 7 to line 9 calculate the corresponding k of trip v_i, and insert the departure location of v_i into $tree_k$. The time complexity of this algorithm is $O(Nlog_M d)$, where N is the total number of trips, M is the maximum fanout of R^*-tree, and d is the number of trips in each time slot.

Figure 6 shows the process of constructing a trip index. Figure 6a shows $tree_1$ is initialized. Figure 6b shows we insert the departure location of a trip into $tree_1$ if the trip's departure time falls into the first time slot. In the same way, we construct the R*-tree for the second time slot, and so on, as Fig. 6c shows.

4 Finding Out Tight Follow-up Trips

Using the trip index, we can find out tight follow-up trips V_i^* of each trip $v_i \in V$ considering both time and spatial constraints. Firstly, we use the time constraint. When a driver finishes v_i at time v_i^{end}, any trip v_j with $v_j^{start} \in [v_i^{end}, v_i^{end} + \delta]$ could be a candidate for v_i's tight follow-up trip. Therefore, we select the time slots $\{t_{i1}, ..., t_{ik}\}$, where t_{i1} is the leftmost time slot containing v_i^{end} and t_{ik} is the rightmost time slot containing $v_i^{end} + \delta$. In other words, the time slot set can cover the interval $[v_i^{end}, v_i^{end} + \delta]$, i.e.,

$$[v_i^{end}, v_i^{end} + \delta] \subseteq t_{i1} \cup t_{i2} \cup \ldots \cup t_{ik}. \quad (3)$$

As Fig. 7 shows, given δ and v_{99}, we select the time slots $\{t_{584}, ..., t_{584+k}\}$, because $v_{99}^{end} \in t_{584}$ and $v_{99}^{end} + \delta \in t_{584+k}$. The time slots $\{t_{584}, ..., t_{584+k}\}$ can cover $[v_{99}^{end}, v_{99}^{end} + \delta]$.

Secondly, we search for candidates in each time slot hit under the spatial constraint. When a driver finishes v_i, she is at v_i^{off}. She can take on the trips with departure locations falling into a small area nearby v_i^{off}. So we do the search by using a spatial range query on the R^*-tree w.r.t. the time slot, as Fig. 7 shows. We will introduce how to designate the area next. In Section 4.1, we designate the area under the assumption that the travel time can be estimated by Euclidean distances. In Section 4.2, our assumption is that the travel time can be estimated by using the traffic data. At last, we identify the real tight follow-up trips and add edges from v_i to these trips.

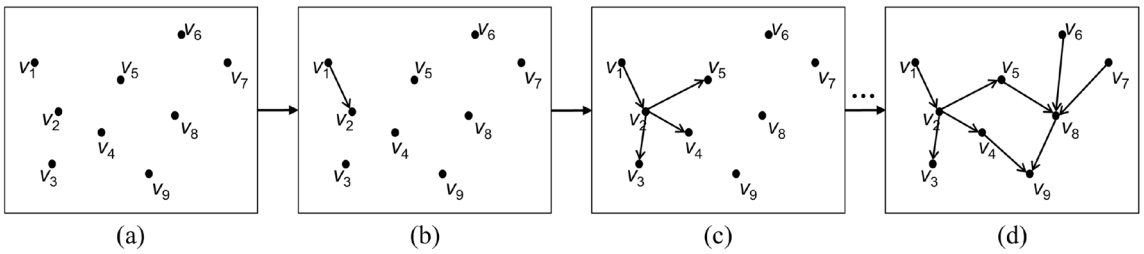


Fig. 8 The process of constructing a trip graph

Algorithm 3: Find Out Tight Follow-up Trips

Input: V: the trip set; $rtree$: R^*-tree forest; δ: free time threshold.
Output: $G(V, E)$: the trip graph.

```
1  Initialize the vertices of G with V;
2  for v_i ∈ V do
3      (t_i1, t_ik) ← GetBoundaryTimeSlots(v_i^end, v_i^end + δ);
4      for t_x ← t_i1 to t_ik do
5          area_x ← GetSearchArea(t_x);
6          V_candid ← RangeQuery(rtree_x, area_x);
           // The rtree_x is the R*-tree w.r.t. t_x.
7      for v_j ∈ V_candid do
8          t_ij ← GetTravelTime(v_i, v_j);
9          if t_ij < δ then
10             E ← E ∪ {e_{v_i→v_j}};
               // v_j is a real tight follow-up trip of v_i.
```

Algorithm 3 summarizes how to find out tight follow-up trips by using R^*-tree forest. Line 3 uses function GetBoundaryTimeSlots() to compute the upper and lower bounds t_{i1} and t_{ik} of the time slots. Then, line 5 uses function GetSearchArea() to calculate the search range corresponding to each time slot. Line 6 takes $area_x$ as the radius of the query range, and performs a spatial range search on the R^*-tree corresponding to the time slot. We can obtain all candidate trips V_{candid} whose departure position is within the query range. Line 7 to line 10 identify the real results from the candidates according to Definition 3. If v_j is a real results, we add edge $e_{v_i \to v_j}$ to the graph G. The time complexity of this algorithm is $O(N(Klog_M d + Z))$, where N is the total number of trips, K is the number of time slots hit, Z is the average number of candidates, M is the maximum fanout of R^*-tree, and d is the number of trips in each time slot.

Figure 8 shows the process of constructing a trip graph. At the beginning, the trip graph only contains nodes which represent different trips $\{v_1, v_2, \ldots, v_9\}$, as Fig. 8a shows. Next, we search for the tight follow-up trips of trip v_1, and v_2 is the result. So, we add a directed edge which is from v_1 to v_2, as Fig. 8b shows. In the same way, we search for the tight follow-up trips of v_2 and the results are v_3, v_4, and v_5. So, we add three directed edges from v_2 to the three trips, as Fig. 8c shows. We search for tight follow-up trips and add directed edges iteratively. After processing the last trip v_9, we obtain the whole trip graph, as Fig. 8d shows.

4.1 Distance-Based Search

In Algorithm 3, designating the search area $area_x$ for $rtree_x$ (w.r.t. time slot t_x) is a key problem. Here we use $t_x^{\vdash}$ to denote t_x's starting time and use $t_x^{\dashv}$ to denote its ending

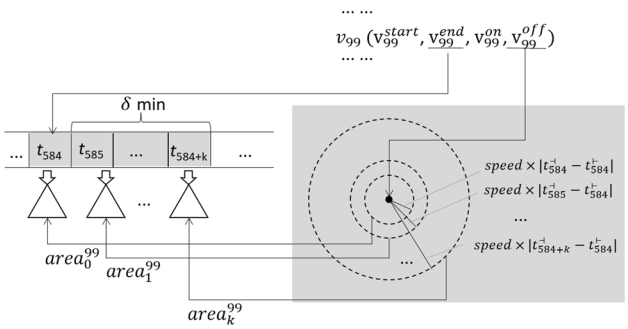


Fig. 9 Distance-based search areas

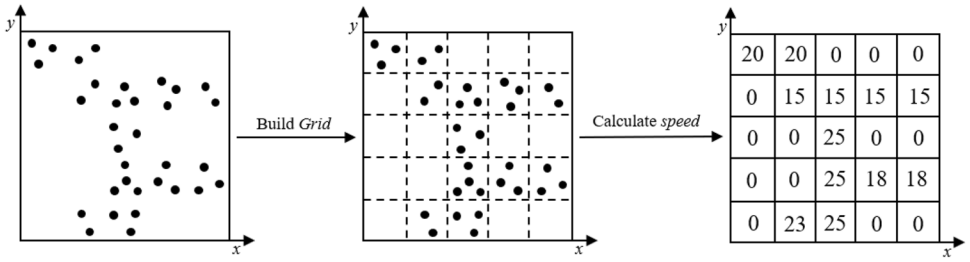


Fig. 10 Build traffic grid

time. We regard a trip v_j to be a candidate, if a driver travels from v_i^{off} and can reach v_j^{on} before $t_x^{\dashv}$. Assume we evaluate the distance between two locations by their Euclidean distance. Thus, $area_x$ could be the disc region of r_x radius from v_i^{off}, where

$$r_x = speed \times |t_x^{\dashv} - t_{i1}^{\vdash}|, \tag{4}$$

and *speed* is the average speed of taxis.

As Fig. 9 shows, to find out tight follow-up trips of v_{99}, we select time slots $\{t_{584}, t_{585}, \ldots, t_{584+k}\}$. Next, for these time slots we must designate corresponding search areas $\{area_0^{99}, area_0^{99}, \ldots, area_k^{99}\}$. These search areas have the same center, namely, v_{99}^{off}, however, they have different radii. For example, the radius of $area_1^{99}$ is $speed \times |t_{584}^{\dashv} - t_{584}^{\vdash}|$, and the radius of $area_2^{99}$ is $speed \times |t_{585}^{\dashv} - t_{584}^{\vdash}|$, and so on. Because a driver can spend more time in taking trips that are in later time slots.

4.2 Traffic-Based Search Areas

In Sect. 4.1, we assume a driver travels at a constant speed, which is an empirical value. However, in real scenarios, the driving speed depends on traffics. This section introduces how to designate search areas considering traffics.

4.2.1 Traffic Grid

Since the driving speed varies in different places, we partition the whole space by a grid C, which consists of $row \times col$ cells. The grid C is called a **traffic grid** where each c_{ij} has a speed c_{ij}^{speed}. To calculate c_{ij}^{speed}, we use the taxi trajectory data. We map trajectory points into cells according to their latitudes and longitudes and calculate the average speed of the trajectory points falling in every cell, i.e.,

$$c_{ij}^{speed} = \frac{\sum_{x=1}^{m} p_x^{speed}}{m}, \tag{5}$$

where m is the number of trajectory points falling in the cell. The p_x's position p_x^{coord} falls into c_{ij} and its speed p_x^{speed} is used to calculate the average speed. Note that GPS can collect a taxi's position together with its speed at each time instant.

Figure 10 illustrates the forming procedure of a traffic grid. The black points indicate trajectory points. After dividing the whole space into a 5×5 grid, the black points fall into different cells. According to Eq. 5, we can obtain the average speed of each cell.

Algorithm 4: Building Traffic Grid

Input: P: the trajectory points; row (or col): the number of rows (or columns) of the grid.
Output: C: the traffic grid.

```
Initialize the row × col grid C;
for each trajectory point p_i ∈ P do
    c_{x,y} ← GetCell(p_i^{coord}, C);
    c_{x,y}^{speed} ← c_{x,y}^{speed} + p_i^{speed};
    c_{x,y}^{m} ← c_{x,y}^{m} + 1;
for each cell c_{i,j} ∈ C do
    c_{i,j}^{speed} ← c_{i,j}^{speed} / c_{i,j}^{m};
```

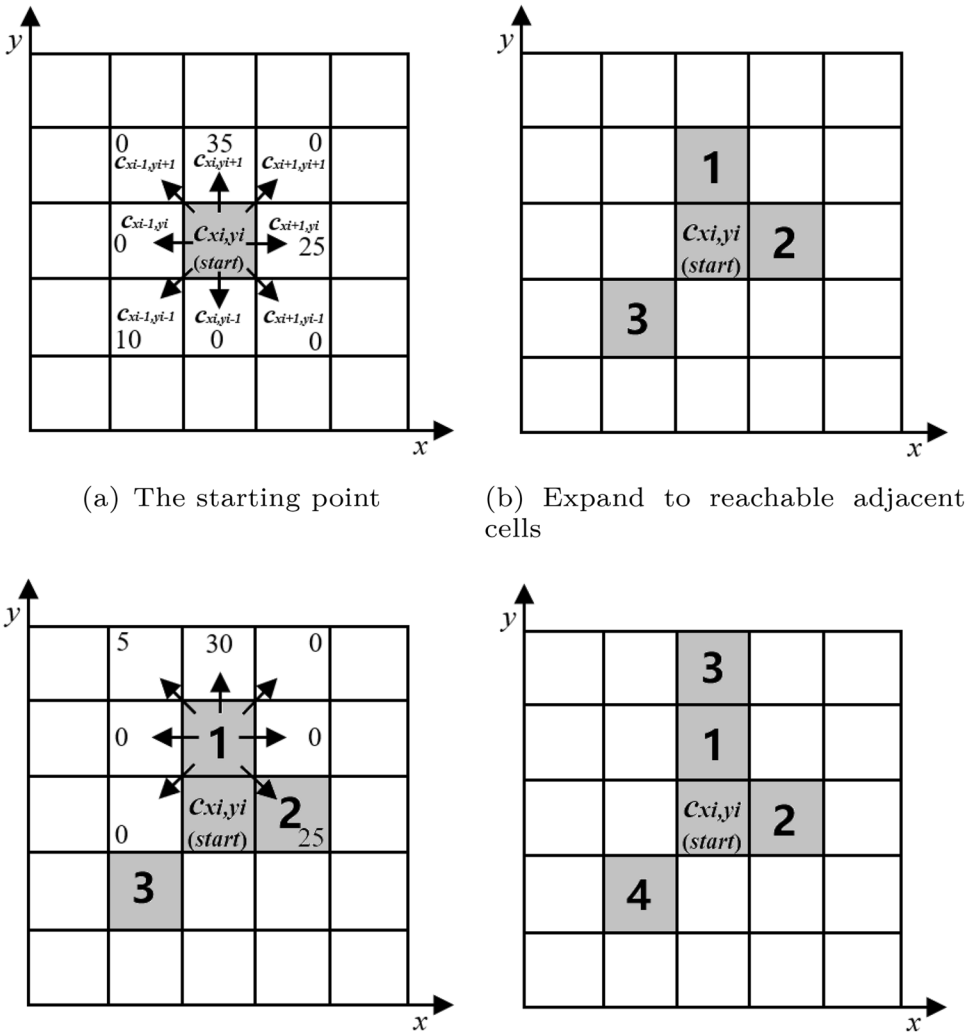


Fig. 11 The process of forming search areas

(a) The starting point

(b) Expand to reachable adjacent cells

(c) Continue to expand

(d) The search areas after expansions

Algorithm 4 describes how to build a traffic grid. In lines 2-5, we map each trajectory point. Line 3 uses function GetCell() to calculate the corresponding cell of p_i according to its latitude and longitude. Line 4 accumulates the sum speed of the trajectory points falling into the cell. Line 5 updates the number of trajectory points falling in the cell. Line 6 and line 7 calculate the average speed of each cell. The traffics can be estimated more accurately if make the cells smaller, however, it is time consuming to build a traffic grid with very small cells.

4.2.2 Getting Search Areas

In the traffic grid C, we find the search areas by expanding cells gradually. The distance between two cells are reduced to the distance between the centers of two cells. The travel time between two cells (from $c_{xi,yi}$ to $c_{xj,yj}$) is

$$travelTime_{c_{xi,yi}\rightarrow c_{xj,yj}} = \frac{1}{2}\times\left(\frac{\|distance_{i,j}\|}{c^{speed}_{xi,yi}}+\frac{\|distance_{i,j}\|}{c^{speed}_{xj,yj}}\right), \tag{6}$$

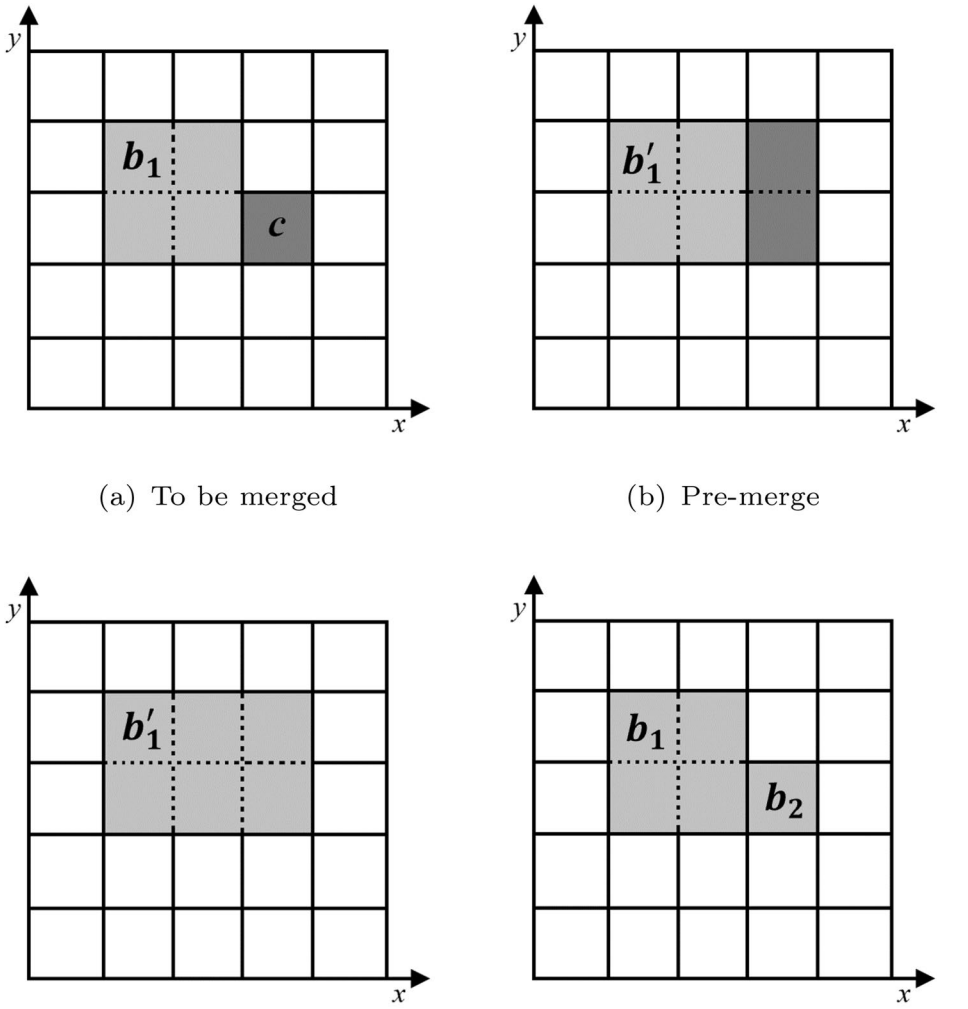
Where $\|distance_{i,j}\|$ denotes the distance from $c_{xi,yi}$ to $c_{xj,yj}$. We take the cell $c_{xi,yi}$ of a departure location v_i^{off} as the starting point of the expansion. We extend the search area to all that are eight cells adjacent to $c_{xi,yi}$, namely, $\{c_{xi-1,yi-1}, c_{xi-1,yi}, \ldots, c_{xi,yi+1}\}$ as Fig. 11a shows. We check whether a cell should be included into the search areas according to the travel time from $c_{xi,yi}$ to this cell. If a driver can reach it within the time threshold δ, we add it into the search areas and continue to check its adjacent cells. The process terminates when the search areas cannot not be expanded.

Figure 11 illustrates the forming process of search areas. The gray cells denote current search areas. Figure 11a shows the starting point of the expansion. The numbers denote the cells' average speeds. Since the travel time from $c_{xi,yi}$ to each cell should be smaller than δ, in Fig. 11b we obtain three reachable cells. Note that the numbers in this figure indicate the priorities of the cells. A cell with a smaller number will be expanded earlier. The priority of a cell depends on the travel time from $c_{xi,yi}$ to the cell. A cell with a shorter travel time can get a higher priority. Figure 11c shows we expand the cell with the highest priority, and Fig. 11d shows the results after this expansion. A new cell is included in the search area and the priorities are changed. Since the expansion order depends on priorities, we use a priority queue to implement the procedure.

Algorithm 5: Get Search Areas

Input: C: the traffic grid, v_i: a trip, δ: free time threshold.
Output: $Areas = \{area_{i1}, \ldots, area_{ik}\}$: the search areas w.r.t. time slots $\{t_{i1}, \ldots, t_{ik}\}$.

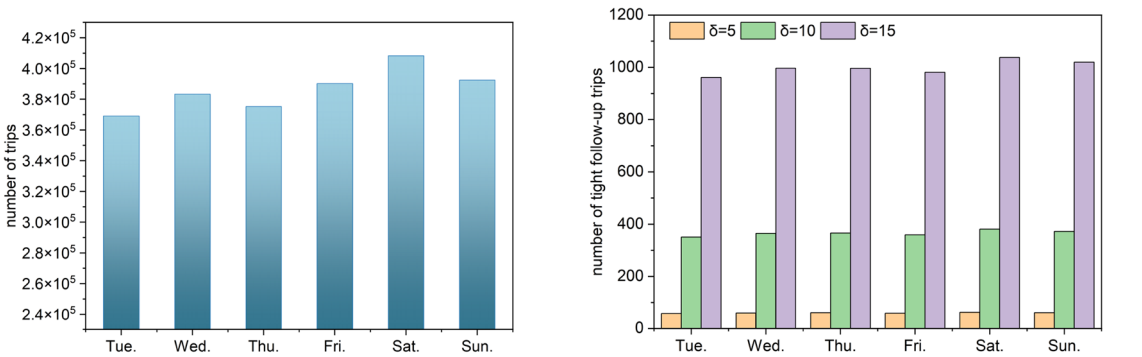
Initialize an array $mtime_{i,j}$ where $i \in 1..row$ and $j \in 1..col$;
// The $mtime_{i,j}$ is used to capture the minimum travel time from $c_{xi,yi}$ to every other cell.
$c_{xi,yi} \leftarrow \mathsf{GetCell}(v_i^{off}, C)$;
Initialize a priority queue Q;
Create a new element elm w.r.t. $c_{xi,yi}$;
Push elm into Q;
$mtime_{xi,yi} \leftarrow 0$; // Set the minimum travel time of $c_{xi,yi}$.
$x \leftarrow i1$; // The x denotes the id of the current time slot.
$max \leftarrow t_x^{\dashv} - t_{i1}^{\vdash}$; // Get the maximum travel time allowed.
while Q *is not empty* **do**
    $top \leftarrow$ Pop the top element of Q;
    if $mtime_{top.id} \leq \delta$ **then**
        // All reachable cells w.r.t. t_x have been collected.
        if $mtime_{top.id} > max$ **then**
            $area_x \leftarrow \mathsf{Merge}(cells)$;
            $x \leftarrow x + 1$; // Move to the next time slot.
            $max \leftarrow t_x^{\dashv} - t_{i1}^{\vdash}$;
            continue;
        else
            $cells \leftarrow cells \cup \{c_{top.id}\}$;
            // Expand search areas.
            for *each adjacent cell* $c_{i,j}$ *of* $c_{top.id}$ **do**
                $newTime \leftarrow mtime_{top.id} + \mathsf{GetTravelTime}(c_{top.id}, c_{i,j})$;
                if $newTime < mtime_{i,j}$ **then**
                    $mtime_{i,j} \leftarrow newTime$;
                    // Update the minimum travel time of $c_{i,j}$.
                    $elm \leftarrow \mathsf{CreateQueElm}(c_{i,j}, mtime_{i,j})$;
                    // The elm's priority depends on $mtime_{i,j}$.
                    Push elm into Q;
    else
        break; // No cell is reachable.



(a) To be merged (b) Pre-merge

(c) When the score meets (d) When the score does not meets

Fig. 12 The procedure of merging cells



(a) Number of trips on different days (b) Number of tight follow-up trips w.r.t. different δ's

Fig. 13 Statistics of taxi trips

Algorithm 5 introduces how to get the search areas of a trip v_i. The algorithm expands search areas in a breadth-first way. Line 1 initializes an array $mtime_{i,j}$, which is used to capture the minimum travel time from $c_{xi,yi}$ to every other cell. Line 2 use function GetCel() to get the cell corresponding to the trip v_i. Line 3 to line 5 initialize a priority queue Q and add $c_{xi,yi}$ to it. Line 6 set the minimum travel time of $c_{xi,yi}$ to 0, because $c_{xi,yi}$ is the departure cell. Line 7 and line 8 initialize x and max, where x denotes the id of the current time slot and max denotes the maximum travel time allowed. Line 10 pop the top element of Q. Line 12 to 16 record the search areas. Line 12 judges whether the record condition is met. Line 13 uses function Merge() to merge the search area, and line 14 and 15 update x and max. Line 18 to line 24 expand the search areas. Line 20 uses function GetTravelTime() to calculate the travel time between $c_{top.id}$ and $c_{i,j}$. If $c_{i,j}$ is reachable, we update the minimum travel time of $c_{i,j}$ and push it into Q. We expand the search areas until there are no more reachable cells.

4.2.3 Merge Cells

The search area consists of a number of reachable cells. A straightforward way is to take each cell as a query range and issue many range queries on R*-tree. But it is time consuming. Another way is to make an MBR (minimum bounding rectangle) of all the cells and take the MBR as the query range. But there will be many unreachable cells included in the query range. On the one hand, we want to reduce the number of range queries, and on the other hand, we want to reduce the number of unreachable cells included in the query range. In this section, we propose a tradeoff method which can obtain a small number of query ranges by merging cells. We merge a cell with its neighboring block if the effect of merging is good enough. The mergeability score is used to evaluate the quality of a new block that will be formed by merging.

Figure 12 graphically depicts the cell merging process. The cells that have been merged into blocks are in light gray. Figure 12a shows the state before merging. At this time, there is a block b_1 and a cell c to be merged, and we try to merge c into the b_1 connected to it. Figure 12b is the state of pre-merging, we expand c, and expect to merge this column of cells into b_1, and calculate the mergeability score. The mergeability score of a block is

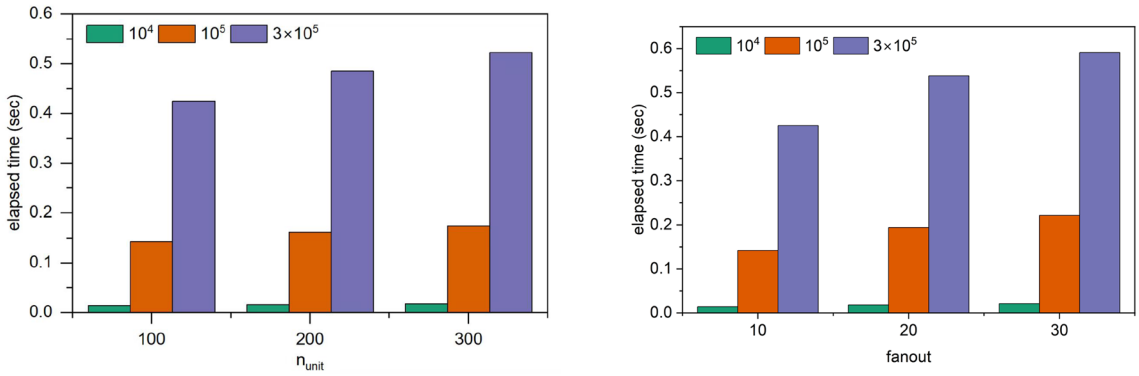
$$score = \frac{m_1 + m_2}{m_{total}} \times \frac{1}{m_3 + 1}, \tag{7}$$

where m_{total} is the total number of cells contained in the block, m_1 is the number of reachable cells, m_2 is the number of blank cells (cells without GPS points), and m_3 is the number of cells that have been merged already. Given a minimum score threshold μ, the block is merged if its mergeability score is larger than μ. Figure 12c shows the new block b_1' formed. When there is more than one block adjacent to cell c, we select the block with the highest mergeability score for merging. When the score is not satisfied, the cell is generated as a new block by itself, as b_2 in Fig. 12d. Note the score should be increased when a cell is determined to be reachable.

Algorithm 6: Merge Cells

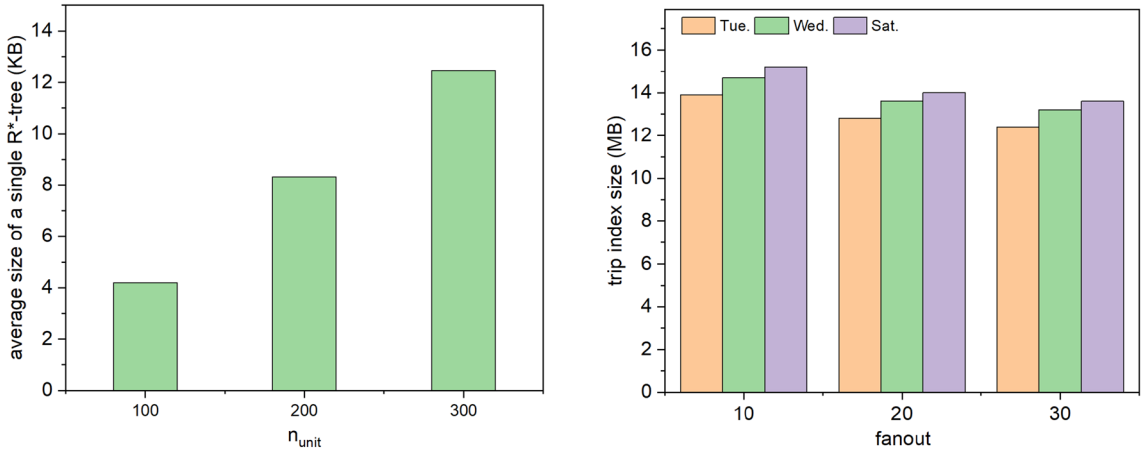
Input: *cells*: the cells to be merged; μ: the minimum mergeability score.
Output: *blocks*: the blocks produced by merging *cells*.

```
1  for each c_{i,j} ∈ cells do
2      if c_{i,j} has not been merged yet then
3          Intialize bestID and bestScore of c_{i,j};
4          for each block b_k adjacent to c_{i,j} do
5              s ← GetMergeScore(b_k, c_{i,j});
6              if s > bestScore then
7                  bestScore ← s;
8                  bestID ← k;
9          if bestScore ≥ μ then
10             MergeOneCell(b_{bestID}, c_{i,j});
11         else
12             blocks ← blocks ∪ {c_{i,j}};
```



(a) Varying n_{unit} (b) Varying fanout of R*-tree

Fig. 14 Elapsed time of constructing trip index



(a) Varying n_{unit} (b) Varying fanout of R*-tree

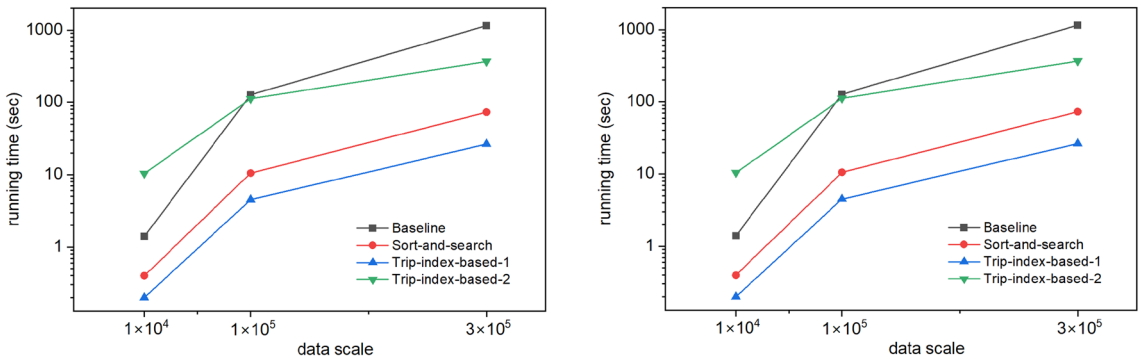
Fig. 15 The index sizes w.r.t. different parameter settings

Algorithm 6 introduces the procedure of merging cells. The inputs of the algorithm are cells to be merged and μ. The outputs are the blocks formed. For each cell to be merged, line 2 determines whether it has been merged. If not, line 3 initializes the best block ID and the best score. Line 4 to line 8 pre-merges each block adjacent to the cell and calculate the mergeability. We keep the best score and the best block ID. Line 5 uses function GetMergeScore() to calculate the score of pre-merging $c_{i,j}$ into b_k. Line 9 checks whether the best score meets the minimum mergeability score. If so, function MergeOneCell() merges $c_{i,j}$ into b_{bestID}. If not, line 12 makes the cell a separate block. When merging, in order to ensure that the block is a rectangle, some irrelevant cells are merged into the block, as shown in Fig. 12b. Each block will records m_1, m_2, m_3, m_{total}, and update them.

This merging problem is in fact to subdivide a rectangular polygon into rectangles [5]. We want to cover all cells with the least rectangles to reduce the number of spatial range queries. But the time complexity of the exact algorithm is high, so we propose this approximation algorithm with a low time complexity.

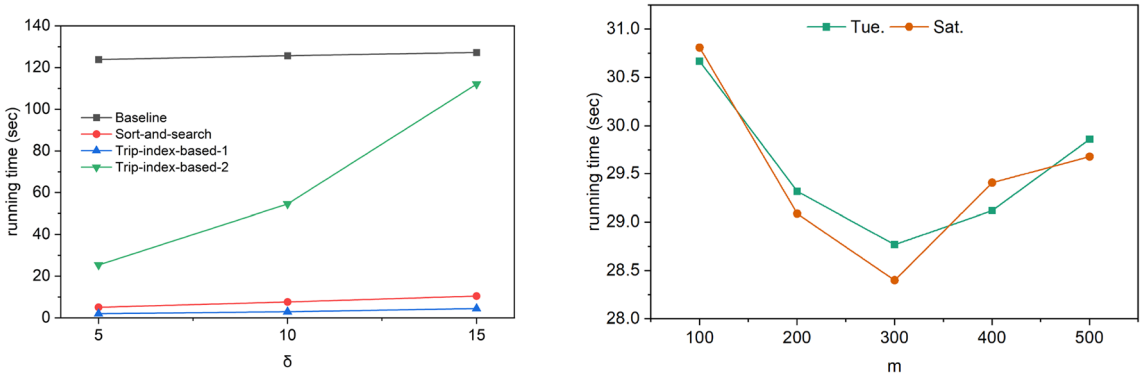
5 Experiment

We conduct experiments to evaluate the performances of the proposed algorithms. We build trip graphs-based on the taxi trajectory data in Shanghai, 2015, and demonstrate the results of minimum fleet analyses by using the trip graphs. All the experiments are executed on a Windows 10 Pro with an Intel(R) Xeon(R) W-2123 3.60GHz CPU and 32GB RAM.



(a) Tuesday (b) Saturday

Fig. 16 Running time of finding out tight follow-up trips



(a) Varying δ (b) Varying m

Fig. 17 Running time when δ or m varies

5.1 Experimental Setting

Datasets The experiments are performed on the taxi trajectory data collected in Shanghai, 2015. We select the trajectory data of one Tuesday, one Wednesday, one Thursday, one Friday, one Saturday, and one Sunday. Each day's dataset contains about 130 million trajectory points. The GPS collects the information of taxi every 5 seconds which includes its position, its speeds, its status (operating, available, suspended), and so on. In the experiments, we extract the taxi ID, the time stamp, the location, and the service status from each trajectory point collected by GPS.

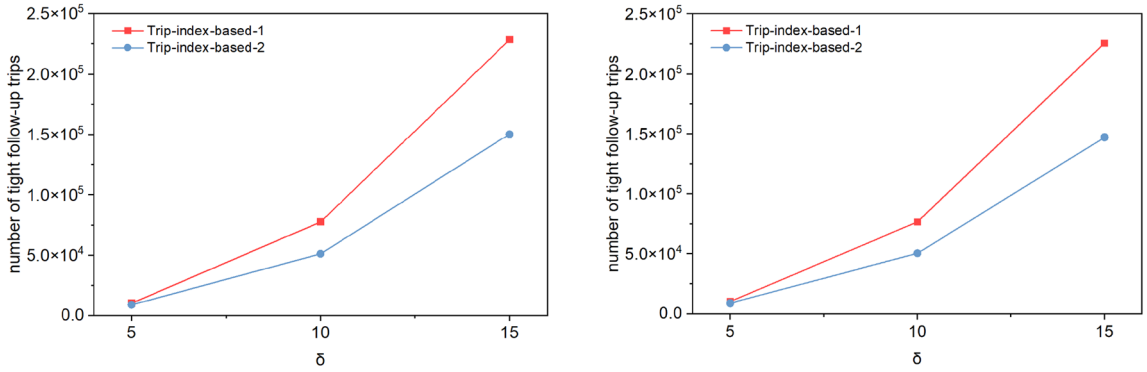
Extract Trips We extract trips from the raw trajectory dataset where the trajectory points are neither grouped by taxi IDs nor sorted by time. In the experiments, we group the trajectory points according to different taxi IDs. For each group we sort the trajectory points in chronological order, and obtain the whole trajectory of a taxi. From the whole trajectory, we extract the trips that are characterized by continuous trajectory points with "operation" status. The "operation" status means the taxi is carrying passengers.

Statistics of Taxi Trips We extract trips from trajectory data on different days. As Fig. 13a shows, the average number of daily trips is 380 thousand. The number of taxi trips on weekdays (i.e., Tuesday, Wednesday, and Thursday) is smaller than the number of taxi trips on weekends (i.e., Friday, Saturday and Sunday). Figure 13b shows the number of tight follow-up trips w.r.t. different δ's. In the experiments, δ is set to 5 minutes, 10 minutes and 15 minutes, respectively. The larger the δ is, the larger the number of tight follow-up trips becomes, because a larger δ means a driver has longer time to drive to the departure location of the next trip.

5.2 Algorithm Performance

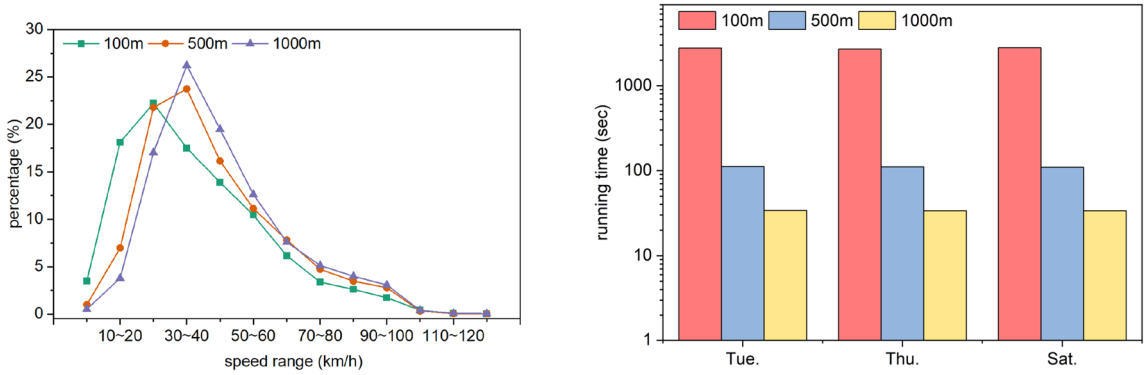
Firstly, we evaluate the performances of the algorithm used to build a trip index. We implement Algorithm 2 in C++ and use the R*-tree source code[3]. We conduct experiments on datasets of different sizes. The datasets contain 10,000,

[3] https://superliminal.com/sources/.



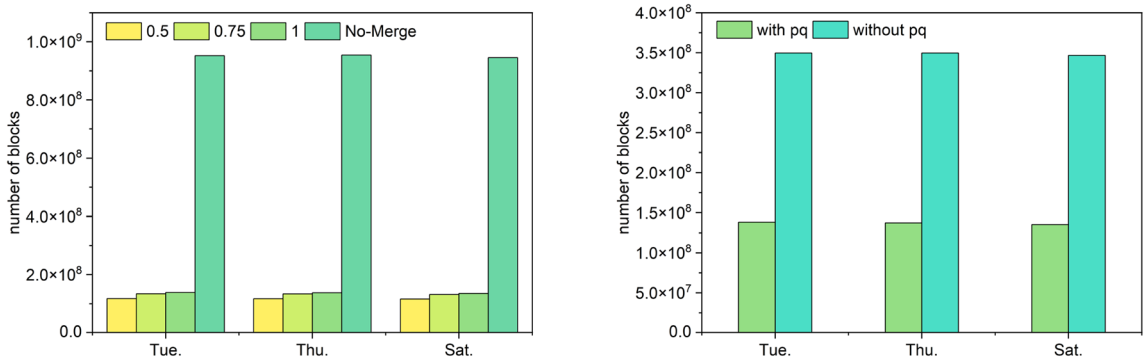
(a) Tuesday (b) Saturday

Fig. 18 Number of tight follow-up trips found



(a) Speed ranges (b) Running time

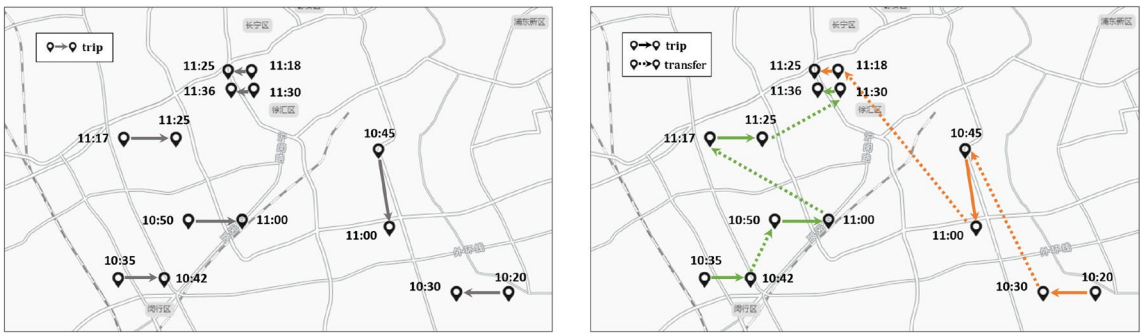
Fig. 19 Performances of trip-index-based-2 w.r.t. different grid sizes



(a) Varying minimum score threshold μ (b) Whether the priority queue is used

Fig. 20 Performances of merging cells (Algorithm 6)

100,000, and 300,000 trips, respectively. As Fig. 14a shows, the parameter m, which is the number of trips in each time slot, is set to 100, 200 and 300, respectively. It takes more time when m is larger. Because a larger m means we should insert more items to the R*-tree. As Fig. 14b shows, the fanout of R*-tree, which is the maximum number of child nodes, is set to 10, 20, and 30, respectively. It takes more time when the fanout is larger. Because a larger fanout



(a) Trips (b) The dispatch of two taxis

Fig. 21 Trips and the dispatch of taxis

makes the tree traverse more child nodes under each node, it takes longer to insert a node, even though the tree is therefore shorter.

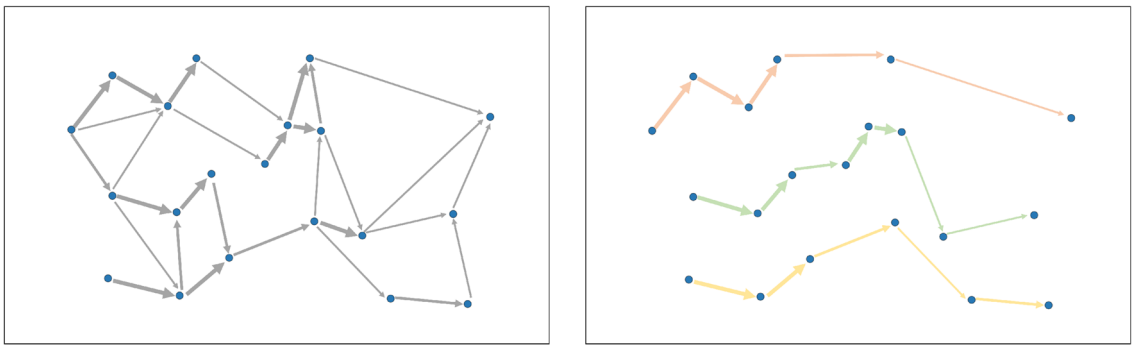
As Fig. 15 shows, we analyze the index sizes under different parameter settings. There are two factors that can influence the index size. One of them is the number of trips in each time slot (i.e., n_{unit}), and the other one is the fanout of R*-tree. Figure 15a shows the average size of a single R^*-tree w.r.t. different n_{unit}'s. The parameter n_{unit} is set to be 100, 200 and 300, respectively, while the fanout of R*-tree keeps in 10. The results show that the R^*-tree is small when n_{unit} is small. Note that we do not show the size of a whole trip index because a trip index is in fact a group of R*-trees and n_{unit} cannot influence the whole size. Different from Fig. 15a and b shows the whole size of a trip index w.r.t. different fanouts of R^*-tree. The fanout of R^*-tree, which is the maximum number of branches, is set to be 10, 20, and 30, respectively, while n_{unit} keeps in 300. The whole index becomes smaller when the fanout grows larger. In this figure, we also show the sizes of indexes for organizing the trips on different days, i.e., Tuesday, Wednesday, and Saturday. Saturday has the largest index for it has the largest number of trips.

Secondly, we evaluate the algorithms for finding out tight follow-up trips. As Fig. 16 shows, we compare four algorithms. The baseline indicates the straightforward algorithm with time complexity $O(n^2)$. The sort-and-search indicates Algorithm 1 which improves baseline. The trip-index-based-1 and trip-index-based-2 indicates the two algorithms supported by the trip index. Both of them conform to the flow of Algorithm 3, while they designate search areas in different ways. The trip-index-based-1 designates search areas considering Euclidean distances (See Sect. 4.1). The trip-index-based-2 designates search areas considering traffics (See Sect. 4.2).

Figure 16 shows the running time of the four algorithms on a weekday (Tuesday, in Fig. 16a) and a weekend (Saturday, in Fig. 16b). The time threshold δ is set to 15 minutes, and the dataset size is set to 10^4, 10^5 and 3×10^5. As shown in the figures, it takes more time to process larger datasets. The running time of baseline grows dramatically, while the running time of the other three algorithms grows relatively smoothly. The trip-index-based-1 runs fastest. The trip-index-based-2 runs slowly, but its results are more realistic since it considers the traffics. It becomes faster than baseline when the dataset size grows to 10^5 and 3×10^5. Because trip-index-based-2 needs to query the traffic grid, when the grid size is set, the running time of a single query is determined. However, with the increase of data scale, the number of trips checked in baseline increases.

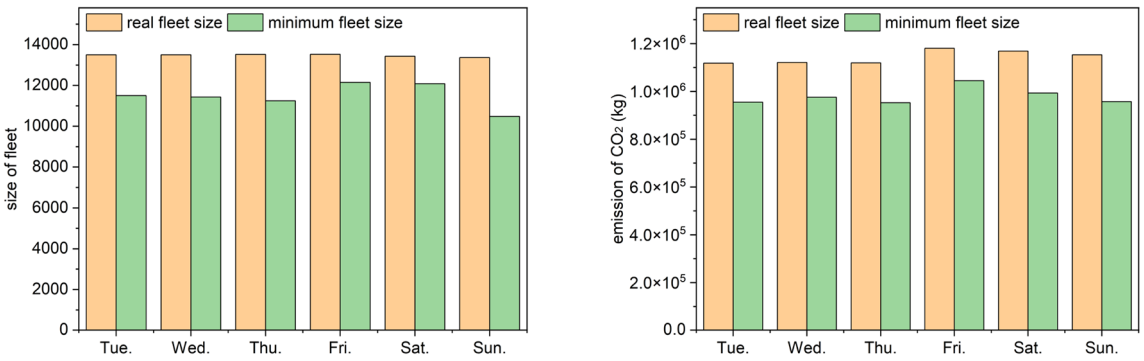
Figure 17 compares the algorithms when the parameter δ or m varies. As Fig. 17a shows, δ is set to 5 minutes, 10 minutes, and 15 minutes, respectively. The δ value has little influence on baseline, however, it has obvious influence on the other three algorithms. When δ grows, their running time increases. Because a larger δ means more trips can be candidates. Fig. 17b shows the running time of trip-index-based-1 w.r.t. to different m's. We use the Tuesday and Saturday datasets. Each of them contains 3×10^5 trips. The m is set to 100, 200, 300, 400 and 500 respectively. The experimental results show that when m is about 300, the algorithm runs fastest.

Figure 18 compares the number of tight follow-up trips found by trip-index-based-1 and trip-index-based-2. We conduct experiments on both the Tuesday dataset (Fig. 18a) and the Saturday dataset (Fig. 18b). The δ is set to 5 minutes, 10 minutes, and 15 minutes, respectively. The number increases when δ gets larger, since a driver has more time to reach departure locations of next trips. In addition, the difference between the number of results found by trip-index-based-1 and the number of results found by trip-index-based-2 also increases when δ gets larger. Because trip-index-based-2 uses the traffic grid, the results are closer to actual situations. When δ increases, the number



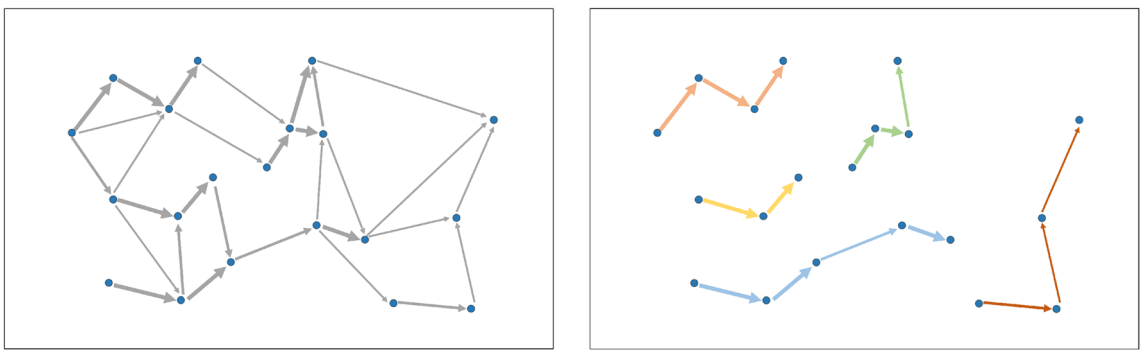
(a) A part of the trip graph in Shanghai (b) The minmum path cover (i.e., three taxis)

Fig. 22 An example of the minimum taxi fleet analysis



(a) Fleet sizes (b) CO_2 emissions

Fig. 23 Analyses of minimum taxi fleets

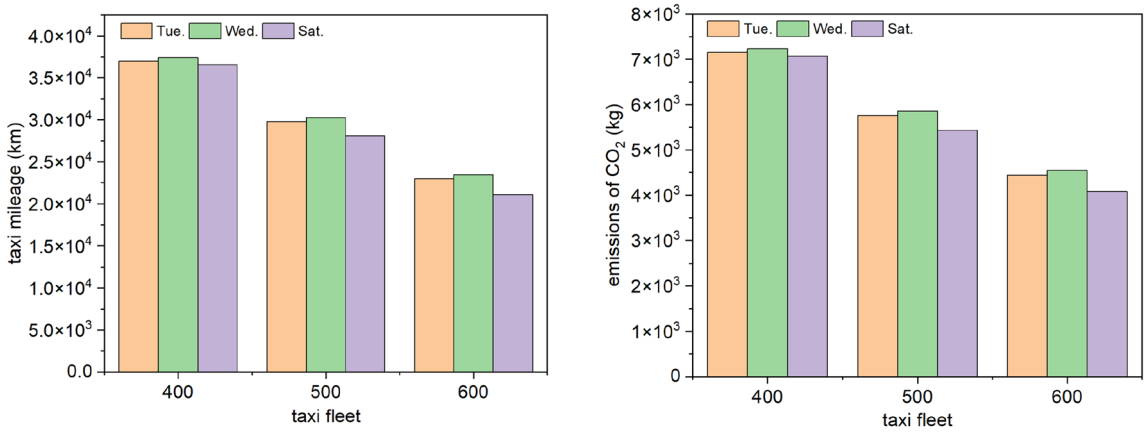


(a) A trip graph with weights (b) An ideal scheduling scheme of five taxis

Fig. 24 An example of the minimum total "idle" mileage analysis

of candidates also increases and the trip-index-based-1 returns much more results.

To observe the statistics of speeds, as Fig. 19a shows, we count the number of cells in different speed intervals. The side length of each cell is set to 1000 meters, 500 meters, and 100 meters, respectively. The x-axis has several speed ranges. The y-axis shows the ratio of the number of cells with a speed range to the total number of cells. The overall distributions of the ratios are similar w.r.t. different cell sizes. However, the uniformity of the speed distribution



(a) Total "idle" mileage (b) CO_2 emissions

Fig. 25 Analyses of minimum total "idle" mileage

varies, and the smaller the cell size, the more uniform the speed distribution. Figure 19 shows the performances of trip-index-based-2 w.r.t. different cell sizes. As Fig. 19b shows, the algorithm runs faster when the cell size is larger, because a larger cell size means fewer cells should be checked.

Figure 20a compares the performances of merging cells w.r.t. different minimum score threshold μ. We set μ to 0.5, 0.75, and 1.0, respectively. The symbol "No-Merge" is the case where no merging is performed. Too many blocks will degrade the performance of trip-index-based-2. Thus, we hope to reduce the blocks as many as possible. As the figure shows, comparing with "No-Merge", the blocks can be reduced by about 85% after merging ($\mu = 1$). It means that the merging operations can reduce the number dramatically. When the threshold μ becomes smaller, the number of blocks decreases. Figure 20b illustrates the role of the priority queue used in trip-index-based-2. The "pq" legend indicates the usage of a priority queue. As the figure shows, the number of blocks can reduce by about 60%, if we implement trip-index-based-2 by using a priority queue. Because the priority queue can make the breadth-first search to be carried out in a way that can improve the merging performance.

5.3 Application Scenario of Trip Graph

Traffic congestion and exhaust pollution are common problems in large cities, however, to restrict the number of taxies blindly may have negative impacts on residents' daily travels. To solve the urban traffic problem, a possible way is to mine the residents' travelling demands and then to estimate how many taxies needed at least. Santi el al. [1] use the trip graph to model residents' travelling demands and solve the minimum fleet problem based on the graph. In fact, the minimum fleet problem is a minimum path cover problem, which is NP-hard and could be solved by using the Hopcroft-Karp algorithm. Another task is to compute an ideal scheduling scheme that can minimize the total "idle" mileage of taxis. Here "idle" means a taxi is travelling without carrying passengers. Given the taxi number and a trip graph, this task is to find out a scheduling scheme that taxis can complete all trips while the total "idle" mileage should be minimized. This task can be converted to a minimum cost flow problem. In this section, we demonstrate both the minimum taxi fleet analysis and the minimum total "idle" mileage analysis based on the trip graph we constructed.

Figure 21 shows the instances of trips and the dispatch of taxis. In Fig. 21a, an arrow with two ballons indicates the beginning and the end of a trip. For simple, we only demonstrate 7 trips in the figure, while in fact the number of trips one day in Shanghai reaches more than 300,000. In Fig. 21b, the two polylines with different colors represent the trips completed by two taxis. A solid arrow indicates a trip. A dotted arrow indicates a driver transfers from one trip to the next trip. The 7 trips in Fig. 21a could be completed by two drivers in Fig. 21b.

Figure 22 shows a trip graph and the minimum path cover on the graph. The trip graph used is a part of the Shanghai's trip graph on one day which consists of 20 trips, as Fig. 22a shows. In the graph, the vertices (i.e., the blue points) denote the trips while the directed edges denote the tight follow-up relationships between two trips. To find out the minimum taxi fleet, we compute the optimal path cover of the graph, as Fig. 22b shows. In the figure, the polylines with different colors illustrate the paths which can cover all the vertices in the graph. In other words, each path indicates a sequence of trips that should be completed by a taxi. Three taxis is enough to complete the 20 trips.

Figure 23 shows the results of the minimum taxi fleet analysis in Shanghai. Figure 23a compares the number of taxis in practice and the number of taxis in the minimum

fleet computed. On different days, the number difference is about 1500. It means that there is still room for the reduction of taxis in Shanghai. Figure 23b compares the CO_2 emissions of real taxi fleets and the minimum fleets on different days. The volumes of CO_2 emissions is computed using the CopertIII model [4]. If the number of taxis in Shanghai is reduced to the minimum number, the volumes of CO_2 emissions can be reduced by nearly 15%.

Figure 24a shows the same trip graph with Fig. 22a. The difference is in this figure the thickness of an edge means its weight. The weight indicates the travel distance between two trips. When a taxi is travelling from the arrival location of the previous trip to the departure location of the next trip, it is in the "idle" state. Fig. 24b shows the ideal scheduling scheme of five taxis. The arrows with the same color represent the order of trips carried by one taxi. Following this scheme, the total "idle" mileage is minimized and thus the volume of CO_2 emissions is also reduced.

Figure 25 shows the results of the minimum total "idle" mileage analyses. Since the algorithm of solving the minimum cost flow problem is time consuming, we extract 1000 trips from 12:00 to 13:00 on Tuesday, Wednesday and Saturday as the experimental datasets. We construct the corresponding trip graphs based on these datasets and compute the ideal scheduling schemes that can minimize the total "idle" mileages. Figure 25a shows the total mileages and Fig. 25b shows the volumes of CO_2 emissions w.r.t. different fleet sizes, i.e., 400, 500 or 600. The experimental results show that when the number of taxis increases, the total mileage and the CO_2 emission are reduced.

6 Related Work

A key point of our work is to construct a trip index which can organize a large amount of trips efficiently. However, most of the existing indexes aim at organizing trajectories and improving the efficiencies of spatial-temporal queries, rather than organizing trips and improving the efficiencies of finding out tight follow-up trips. The queries on trajectories include k nearest neighbor (kNN) queries, range queries, path queries and so on [6, 7]. To support these queries, 3D-Rtree [8] considers the timestamps of trajectory points as well as their spatial coordinates, when organizing the points by an R*-tree. STR-tree [9] takes the identifications of trajectories into account. It modifies the insertion and the split strategies of tree nodes. MV3R-tree [10] has good performances for time interval queries. It combines two structures and has small auxiliary 3D-Rtrees built on the leaf nodes of the multiversion R-tree. CSE-tree [11] is a probabilistic model that simulates user behaviors of uploading trajectories. CSR-tree, which is a variation of CSE-tree, combines B^+-tree and dynamic arrays. It has a smaller size and lower updating costs. GAT [12] is a novel hybrid grid index, which can organize the trajectories and activities hierarchically. PPQ-trajectory [13] aims at improving the performances of queries on massive dynamic trajectories by using spatio-temporal quantization methods. To improve the memory throughput and to reduce CPU-cache misses, Wang et al. propose an in-memory column-oriented trajectory storage [14]. They divide the database into parts, where trajectory points collected at the same moment are stored together and aligned in main memory. TrajStore [15] is a dynamic storage system which let the storage change with the index. TrajStore maintains an optimal index which can answer incoming queries. TrajMesa [16] is a holistic distributed NoSQL trajectory storage engine. It adopts a distributed storage mode, reduces the storage size, and designs a pruning strategy. Fang et al. propose a multi-source deep traffic prediction framework over spatio-temporal trajectory data [17]. Fang et al. propose ST2Vec [18], a representation learning based solution that considers fine-grained spatial and temporal relations between trajectories. Lan et al. propose VRE [19], which is a system used to manage trajectory data in a versatile, robust, and economical manner. VRE can support various queries and multiple distance functions. Our problem is different from the problems they focus on.

The trip graph we constructed can be used to solve the minimum taxi fleet problem [2]. Comparing the minimum number of taxis with the real number of taxis one day, taxi operation companies can further optimize the vehicle scheduling algorithm [1, 20]. Governments can further evaluate the impact of the number of taxis on environments [3] by using the exhaust emission model [4]. In addition, researchers also study the fleet management problems in the real time scenarios. Jin et al. regard the taxi scheduling as a parallel sorting problem and make decisions hierarchically [21]. Zhang et al. propose a method to match multiple order pairs simultaneously within a short time window [22]. Seow et al. solve the multi-agent taxi scheduling problem in local areas [23]. Xu et al. propose a reinforcement learning method to optimize vehicle utilizations and user satisfactions in a global perspective [24]. Considering the uncertainties of future orders, Wei et al. propose a reinforcement learning method that can make forward-looking decisions in order to improve the service qualities [25]. Lin et al. propose a context-multi-agent and actor-critic framework to capture random supplies and demand variations [26]. Xu et al. propose a recommender system which guides drivers to better locations with more passengers [27]. Li et al. propose a fleet manager that brings agents closer to resources [28]. The fleet manager uses a dynamic weighting method to send agents (i.e., the taxi drivers) to locations where more resources (i.e., passengers) may show up. Ming et al. propose a collaborative spatio-temporal searching method for fleet managements [29]. Wang et al. propose a citywide and real-time

model for estimating the travel time [30]. They use a three dimension tensor to model drivers' travel time, and find the estimated optimal trajectories by using a dynamic programming solution.

7 Conclusions

In this paper, we propose a trip index that can organize daily trips in big cities. Supported by the trip index, we propose search algorithms which can find out the tight follow-up trips fast. We construct the trip graph by taking the trips as vertices and connecting the vertices if the tight follow-up relationships exist. The trip graph can be used to solve various transportation problems including the minimum fleet problem. The experimental results show that the proposed algorithms can construct trip graphs faster than the straightforward methods. We also demonstrate the analyses of the minimum taxi fleet in Shanghai which is a typical application scenarios of the trip graphs constructed.

Author Contributions HY proposed the follow-up trip searching algorithms, implemented the index constructing algorithm and the searching algorithms, and conducted the experiments on real datasets. XG defined the problems, designed the trip index, and polished the index constructing algorithm and the follow-up trip searching algorithms. XL provided the taxi trajectory dataset, and designed the experiments about the application scenarios of trip graphs. WB analyzed the experimental results about the application scenarios, and revised the typos in the manuscript. TZ made suggestions on how to improve the trip index and the search algorithms, and polished the writing of the manuscript.

Funding National Natural Science Foundation of China (61602031), Xi Guo Shanghai Natural Science Foundation (21ZR1466600), Xiao Luo Fundamental Research Funds for the Central Universities (FRF-IDRY-19-023), Xi Guo Fundamental Research Funds for the Central Universities (2022-5-YB-03), Xiao Luo.

Data Availability Source codes and datasets: https://github.com/clover-Saber/Construct-Trip-Graphs-by-Using-Taxi-Trajectory-Data.git (1) The source codes of constructing a trip index; (2) The source codes of searching algorithms; (3) The source codes of minimum fleet analyses; (4) The trip graph including about 150,000 vertices and 100,000,000 edges. (Due to the limitations on the usages of the real dataset, we removed the geographical coordinates of trips).

Declarations

Conflict of interest The authors declare that they have no conflict of interests.

References

1. Santi P, Resta G, Szell M, Sobolevsky S, Strogatz SH, Ratti C (2014) Quantifying the benefits of vehicle pooling with shareability networks. Proceed Nat Acad Sci 111(37):13290–13294
2. Vazifeh MM, Santi P, Resta G, Strogatz SH, Ratti C (2018) Addressing the minimum fleet problem in on-demand urban mobility. Nature 557(7706):534–538
3. Yan L, Luo X, Zhu R, Santi P, Wang H, Wang D, Zhang S, Ratti C (2020) Quantifying and analyzing traffic emission reductions from ridesharing: a case study of shanghai. Transp Res Part D: Trans Environ 89:102629
4. Ntziachristos L, Samaras Z, Eggleston S, Gorissen N, Hassel D, Hickman A et al. (2000) Copert iii. Computer Programme to calculate emissions from road transport, methodology and emission factors (version 2.1), European Energy Agency (EEA), Copenhagen
5. Eppstein D (2009) Graph-theoretic solutions to computational geometry problems. In: International workshop on graph-theoretic concepts in computer science, Springer, pp. 1–16
6. Zheng Y (2015) Trajectory data mining: an overview. ACM Trans Intell Sys Technol (TIST) 6(3):1–41
7. Wang S, Bao Z, Culpepper JS, Cong G (2021) A survey on trajectory data management, analytics, and learning. ACM Comp Surv (CSUR) 54(2):1–36
8. Zhu Q, Gong J, Zhang Y (2007) An efficient 3d r-tree spatial index method for virtual geographic environments. ISPRS J Photogr Remote Sens 62(3):217–224
9. Pfoser D, Jensen C.S, Theodoridis Y., et al. (2000) Novel approaches to the indexing of moving object trajectories. In: VLDB, pp. 395–406
10. Tao Y, Papadias D (2001) The mv3r-tree: a spatio-temporal access method for timestamp and interval queries. In: Proceedings of Very Large Data Bases Conference (VLDB), 11–14 September, Rome
11. Wang L, Zheng Y, Xie X, Ma W.-Y (2008) A flexible spatio-temporal indexing scheme for large-scale gps track retrieval. In: The Ninth International Conference on Mobile Data Management (mdm 2008), IEEE, pp. 1–8
12. Zheng K, Shang S, Yuan N.J, Yang Y (2013) Towards efficient search for activity trajectories. In: 2013 IEEE 29Th International conference on data engineering (ICDE), IEEE, pp. 230–241
13. Wang S, Ferhatosmanoglu H (2020) Ppq-trajectory: spatio-temporal quantization for querying in large trajectory repositories. arXiv preprint arXiv:2010.13721
14. Wang H, Zheng K, Xu J, Zheng B, Zhou X, Sadiq S (2014) Sharkdb: an in-memory column-oriented trajectory storage. In: Proceedings of the 23rd ACM International conference on conference on information and knowledge management, pp. 1409–1418
15. Cudre-Mauroux P, Wu E, Madden S (2010) Trajstore: an adaptive storage system for very large trajectory data sets. In: 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010), IEEE, pp. 109–120
16. Li R, He H, Wang R, Ruan S, Sui Y, Bao J, Zheng Y (2020) Trajmesa: A distributed nosql storage engine for big trajectory data.

In: 2020 IEEE 36th International conference on data engineering (ICDE), IEEE pp. 2002–2005
17. Fang Z, Pan L, Chen L, Du Y, Gao Y (2021) Mdtp: a multi-source deep traffic prediction framework over spatio-temporal trajectory data. Proceed. VLDB Endowm. 14(8):1289–1297
18. Fang Z, Du Y, Zhu X, Hu D, Chen L, Gao Y, Jensen CS (2022) Spatio-temporal trajectory similarity learning in road networks. In: Proceedings of the 28th ACM SIGKDD conference on knowledge discovery and data mining, pp. 347–356
19. Lan H, Xie J, Bao Z, Li F, Tian W, Wang F, Wang S, Zhang A (2022) Vre: a versatile, robust, and economical trajectory data system. Proceed VLDB Endowm 15(12):3398–3410
20. Alonso-Mora J, Samaranayake S, Wallar A, Frazzoli E, Rus D (2017) On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. Proceed Nat Acad Sci 114(3):462–467
21. Jin J, Zhou M, Zhang W, Li M, Guo Z, Qin Z, Jiao Y, Tang X, Wang C, Wang J., et al. (2019) Coride: joint order dispatching and fleet management for multi-scale ride-hailing platforms. In: Proceedings of the 28th ACM international conference on information and knowledge management, pp. 1983–1992
22. Zhang L, Hu T, Min Y, Wu G, Zhang J, Feng P, Gong P, Ye J (2017) A taxi order dispatch model based on combinatorial optimization. In: Proceedings of the 23rd ACM SIGKDD International conference on knowledge discovery and data mining, pp. 2151–2159
23. Seow KT, Dang NH, Lee D-H (2009) A collaborative multiagent taxi-dispatch system. IEEE Trans Autom Sci Eng 7(3):607–616
24. Xu Z, Li Z, Guan Q, Zhang D, Li Q, Nan J, Liu C, Bian W, Ye J (2018) Large-scale order dispatch in on-demand ride-hailing platforms: a learning and planning approach. In: Proceedings of the 24th ACM SIGKDD international conference on knowledge Discovery & data mining, pp. 905–913
25. Wei C, Wang Y, Yan X, Shao C (2017) Look-ahead insertion policy for a shared-taxi system based on reinforcement learning. IEEE Access 6:5716–5726
26. Lin K, Zhao R, Xu Z, Zhou J (2018) Efficient large-scale fleet management via multi-agent deep reinforcement learning. In: Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining, pp. 1774–1783
27. Xu Z, Men C, Li P, Jin B, Li G, Yang Y, Liu C, Wang B, Qie X (2009) When recommender systems meet fleet management: practical study in online driver repositioning system. In: Proceedings of the web conference 2020, pp. 2220–2229
28. Li W (2020) A fleet manager that brings agents closer to resources: Gis cup. In: Proceedings of the 28th International conference on advances in geographic information systems, pp. 655–658
29. Ming L, Hu Q, Dong M, Zheng B (2020) An effective fleet management strategy for collaborative spatio-temporal searching: Gis cup. In: Proceedings of the 28th International conference on advances in geographic information systems, pp. 651–654
30. Wang Y, Zheng Y, Xue Y (2014) Travel time estimation of a path using sparse trajectories. In: Proceedings of the 20th ACM SIGKDD international conference on knowledge discovery and data mining, pp. 25–34