The VLDB Journal
https://doi.org/10.1007/s00778-018-0499-4

REGULAR PAPER

Go slow to go fast: minimal on-road time route scheduling with parking facilities using historical trajectory

Lei Li[1] · Kai Zheng[2] · Sibo Wang[1] · Wen Hua[1] · Xiaofang Zhou[1,3]

Received: 15 October 2017 / Revised: 22 January 2018 / Accepted: 28 February 2018

Abstract
For thousands of years, people have been innovating new technologies to make their travel faster, the latest of which is GPS technology that is used by millions of drivers every day. The routes recommended by a GPS device are computed by path planning algorithms (e.g., fastest path algorithm), which aim to minimize a certain objective function (e.g., travel time) under the current traffic condition. When the objective is to arrive the destination as early as possible, waiting during travel is not an option as it will only increase the total travel time due to the *First-In-First-Out* property of most road networks. However, some businesses such as logistics companies are more interested in optimizing the actual on-road time of their vehicles (i.e., while the engine is running) since it is directly related to the operational cost. At the same time, the drivers' trajectories, which can reveal the traffic conditions on the roads, are also collected by various service providers. Compared to the existing speed profile generation methods, which mainly rely on traffic monitor systems, the trajectory-based method can cover a much larger space and is much cheaper and flexible to obtain. This paper proposes a system, which has an online component and an offline component, to solve the minimal on-road time problem using the trajectories. The online query answering component studies how parking facilities along the route can be leveraged to avoid predicted traffic jam and eventually reduce the drivers' on-road time, while the offline component solves how to generate speed profiles of a road network from historical trajectories. The challenging part of the routing problem of the online component lies in the computational complexity when determining if it is beneficial to wait on specific parking places and the time of waiting to maximize the benefit. To cope with this challenging problem, we propose two efficient algorithms using *minimum on-road travel cost function* to answer the query. We further introduce several approximation methods to speed up the query answering, with an error bound guaranteed. The offline speed profile generation component makes use of historical trajectories to provide the traveling time for the online component. Extensive experiments show that our method is more efficient and accurate than baseline approaches extended from the existing path planning algorithms, and our speed profile is accurate and space efficient.

Keywords Road network · Shortest path · Trajectory

1 Introduction

With the prevalence of GPS enabled devices and wireless network, various of navigation systems have been widely adopted by public transportation systems, logistics companies, private vehicles and a broad range of location-based services. These systems first find where we are on planet earth, then compute a reasonable path to our destinations, most of which are based on shortest path algorithms [1–3].

✉ Kai Zheng
zhengkai@uestc.edu.cn

✉ Xiaofang Zhou
zxf@itee.uq.edu.au

Lei Li
l.li3@uq.edu.au

Sibo Wang
sibo.wang@uq.edu.au

Wen Hua
w.hua@uq.edu.au

[1] School of ITEE, University of Queensland, Brisbane, QLD 4072, Australia

[2] School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China

[3] School of Computer Science and Technology, Soochow University, Suzhou, China

Published online: 21 March 2018

During the trip, they provide turn-by-turn navigation using real-time map-matching and real-time path computation. Some of them even keep users' trajectories, like O2O taxi service providers *Uber* and *DiDi*. In fact, they are becoming more and more popular around the world and have obtained a tremendous amount of trajectories generated by their taxi drivers every day. However, although these trajectories can reveal the traffic conditions of different parts of a city at different time periods, they are mostly used for behavior analysis and customer support.

In spite of their popularity, there are still some untreated shortcomings. First of all, the paths they generated are mostly based on distance rather than actual travel time [1–3]. Obviously, shortest path does not necessarily have the shortest travel time. Thus, it cannot satisfy many users' needs, i.e., arriving the destination earlier. Furthermore, it may even lead many cars to traffic jams during the rush hour.

Secondly, even though the travel time is considered in some path planning algorithms [4–9], they still do not allow waiting during the trip. The common optimization goal of them is the *total travel time*, which is the difference between the departure time and arrival time, and is made up of the on-road time and waiting time. In a time-dependent road network where the cost associated with road segment can change over time, the existing path planning problems make use of an important observation known as the *FIFO* property, which means a vehicle enters a road segment first will also reach the end of road segment first in spite of the time-dependent nature [10]. So for an *FIFO* road network, there is no need to consider waiting during travel since waiting can only increase the total time. However, for many users such as logistics companies with heavy trucks, the actual on-road time (i.e., the time when the engine is running) becomes critical as it directly relates to the fuel consumption which can be as high as 80% of their operational cost. As long as the goods can be delivered on time, reducing the actual on-road time can be more economic than arriving the destination earlier. On the other hand, tourists would also like to reduce their time spent on road so that they can spend more time on the tourist attractions. On a bigger view, the more cars that reduce their on-road time, the better traffic condition there would be, which would lead to less exhausted emission and a better environment. This motivates us to study a new kind of path planning algorithm that optimizes the on-road time by waiting strategically in certain places along the route to avoid predictable traffic jams. To better understand how waiting can shorten the on-road time when traveling, consider a road network with five vertices as shown in Fig. 1. Three of them are ordinary vertices, and two of them are parking vertices that allow waiting. The traveling cost functions are as shown in Fig. 1b–f. These linear functions simulate the speed profile we generate from trajectories. We choose linear function rather higher-order ones because it suffers less from overfitting and it is easy to handle. In fact, most works in this field use linear function. Suppose the starting time from v_1 is 0 and the latest arrival time at v_5 is 130. The fastest path takes 105 time units ($v_1 \rightarrow v_2 : 40$; $v_2 \rightarrow v_3 : 70$; $v_3 \rightarrow v_5 : 105$), and its on-road travel time is also 105. However, if we still start from v_1 at 0 and arrive v_2 at 40, but travel from v_2 to v_4 and arrive v_4 at 95, the current on-road time is 95. Then, we wait on v_4 and depart on 120, the cost from v_4 to v_5 reduces to 5. So the on-road travel time of this path is 100. So by taking advantages of these parking vertices, we can obtain a route that has shorter on-road travel time. More application scenarios are presented in Sect. 4.4 after the algorithm is fully described.

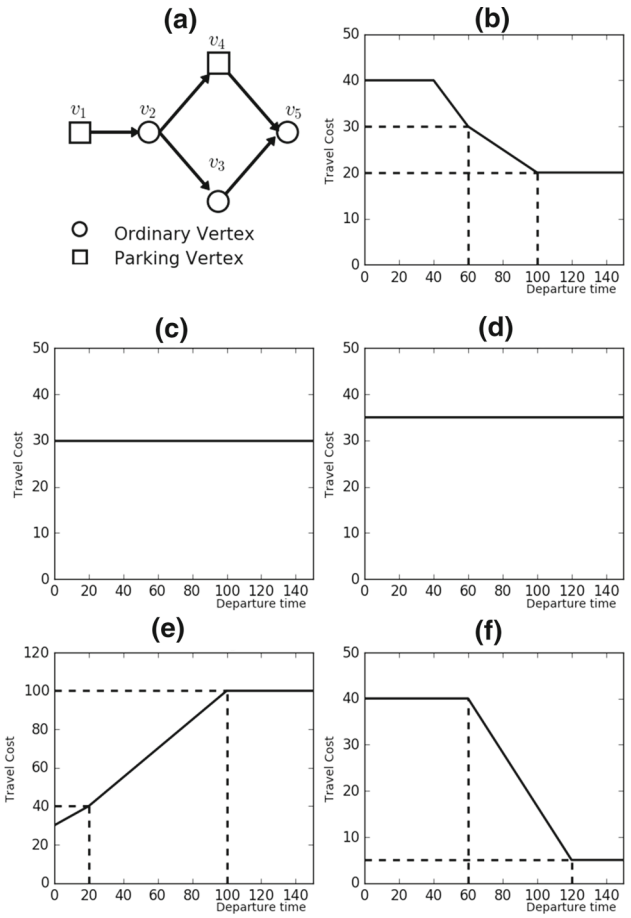

Fig. 1 A road networking with parking vertices (**a**) and the corresponding time-dependent weight for each edge over time domain (0–150) (**b**)–(**f**)

Last but not least, it is hard to obtain the travel time information of the roads. Currently, most of the systems depend on sensors deployed throughout the city [11]. Apparently, such approach is accurate but expensive to cover the city center, and impossible to cover the entire road network. However, with the vast amount of historical and real-time trajectory data at hand, we are able to derive the travel time information at a much larger scale with little cost.

We distinguish the term *path* and *route* by if they are influenced by time or not. *Path* is used in the scenarios when no waiting is allowed after departure. Therefore, a *path* is just a series of vertices. Once the departure time from the source vertex is fixed, the arrival time at the destination vertex is fixed as well. *Route* is used when some of the vertices allow waiting. Therefore, we also need to provide corresponding waiting time or departure time on these waiting vertices. So a route is a path plus the departure time of each vertex.

In this work, we model a road network as a time-dependent graph, where each edge is associated with a function that returns the time cost of traveling the edge for a given departure time from the starting vertex, and these functions are generated from the trajectory database. There are two types of vertices in this graph: *ordinary vertices* that do not allow waiting, and *parking vertices* that do. This model considers the phenomenon that some vehicles may choose to stop at some places to avoid traffic jams. The proposed query, *minimal on-road time route query* (*MORT*), aims to find a route that consists of not only a consecutive of edges in the road network, but also a *waiting plan* that determines the amount of time to stop at a parking vertex in order to minimize on-road time. So it is actually a route scheduling algorithm rather than a path planning problem. This is different from the previous problems that aim at minimizing the *total travel time*, which includes both the on-road time and waiting time. Clearly, a *MORT* query is more complicated than traditional path planning queries that minimize the total travel time. First of all, it needs to decide whether waiting at certain parking vertices, or even taking a detour to a parking vertex, can save on-road time at all. Secondly, if waiting on this parking vertex has benefit, it needs to further determine the waiting time on it. Finally, because waiting on any vertex is allowed, the graph that *MORT* query runs on does not need to follow *FIFO* property, which is the basis of all the existing algorithms.

In fact, the existing path planning algorithms cannot solve this problem even under *FIFO* setup. First of all, the *shortest path algorithms* [1–3,12] only work on static edge weights. Thus, it cannot handle the time-dependent costs because the time-dependent one has many different optimal solutions during a time interval. Secondly, the *single starting-time fastest path (SSFP)* algorithm [10] does not allow waiting at any vertex. Even though it has the ability to cope with time-dependent costs, it cannot solve our problem. Finally, the *interval starting-time fastest path (ISFP)* algorithms [4,5] allow waiting on the starting vertex, but they do not allow waiting on the intermediate vertices since it would simply result in a longer total travel time. One naive approach to find an approximate *MORT* route based on *ISFP* algorithms is to select the optimal waiting time on each parking vertex along the path in a greedy fashion. Firstly, it runs *ISFP* algorithm on the starting vertex to get the optimal departure time t_s^* on starting vertex v_s. Then, it runs *ISFP* algorithm on the first parking vertex v_{p1} along the path with its arrival time from v_s at time t_s^* as the starting time, and gets the optimal departure time t_{p1}^* from v_{p1}. After that, it runs the *ISFP* on the first parking vertex along the new path from v_{p1} again to get its optimal departure time. The procedure runs iteratively until the destination vertex is reached. However, this approach has two problems: Obviously, it runs *ISFP* multiple times, so its computation time is long. A more serious problem is that this approach has no guarantee to find the optimal solution at all as it is a greedy method with no backtracking (the first parking site on a route is just an accidental stop point "from" a path that has not considered parking as an optimization option). Figure 2 shows an real-life example of the comparison of our algorithm, *ISFP* [5] and the naive iterative approach. The example illustrates paths from location $A(31.2414, 121.304)$ to $B(31.2559, 121.386)$ in Shanghai, whose shortest distance is 10 km. The starting time interval is set from 10:00 to 16:00 and the latest arrival time is 19:00. *ISFP* finds a path with an on-road travel time of 1385 s, iterative approach finds a path of 1130 s, while our algorithm finds a route of 986 s.

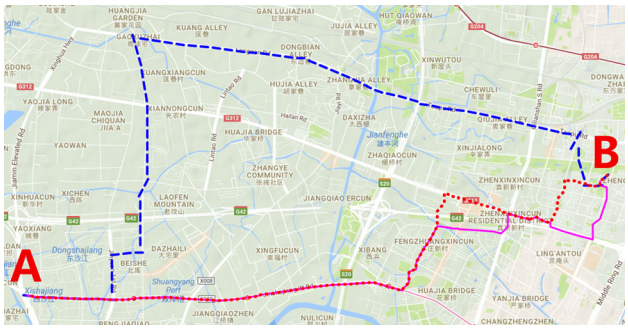

Fig. 2 *MORT* route (red dotted), fastest path (blue dashed) and recursive fastest path (pink solid)

In this paper, we present a system to answer the minimal on-road travel route query, as well as all the other existing time-dependent path queries, using speed profiles generated from trajectories. The system has an online query answering component and an offline speed profile generation component. The *Online Component* has two algorithms to find the minimal on-road travel route accurately and an approximation algorithm to answer query faster with error bounded. Both of the accurate algorithms construct and maintain a set of *Minimum Cost Functions* to record the minimal on-road time from the starting vertex to the other vertices at different arrival times. The first algorithm builds the minimum cost functions over the whole query time interval iteratively in a *Dijkstra* way, while the second algorithm constructs it sub-time-interval by sub-time-interval instead. We observe a *non-increasing property* for the parking vertices, which integrates the waiting time benefit into the minimum cost function. Both of them support user specifying different min-

imum waiting times when waiting on parking vertices. We also provide a route retrieval solution to return routing schedule satisfying user's requirement on the arrival time. It is worth noting that our *MORT* algorithm is more general than the existing time-dependent path algorithms. First of all, if we treat the parking vertices as normal vertices, our algorithm can solve the *ISFP* problem. Moreover, if we further prohibit waiting on starting vertex, our algorithm can solve the *SSFP* problem. In fact, both *ISFP* and *SSFP* are the special cases of *MORT*. Furthermore, in order to speed up the query answering time, we propose an α-*MORT* approach to provide approximate result by pruning some of the turning points in each vertex's minimum cost function. Because the error grows exponentially as the route expands, we have to view the error bound as a pruning power budget and distribute it along the route. We propose three ways to achieve it: *Even Distribution*, *Exponential Distribution* and *Dynamic Exponential Distribution*. The *Offline Component* reads the raw trajectories from the database and then generates a reliable speed profile from these trajectories by map matching, speed data collection, missing value estimation and compression.

In summary, our contributions are listed as follows:

- We propose a system to answer a general form of time-dependent route scheduling problem *MORT* using the historical trajectory.
- The online query answering component solves *MORT* problem, which makes use of parking facilities in a road network to minimize the on-road travel time, instead of the total travel time. We propose a *minimum cost function* and two novel algorithms to solve the *MORT* route scheduling problem efficiently and accurately, and an approximation approach for faster query answering. Our algorithms can handle real-life road network with dynamic and complex speed profiles. Both of them are able to address other existing types of time-dependent path planning problems if no parking vertices are considered.
- The *Basic MORT algorithm* performs the *MORT* search for a vertex after each iteration, until the destination is reached. We show that its time complexity is $O(T|V|\log|V|+T^2|E|)$. The *Incremental MORT algorithm* visits the vertices starting from a small subinterval to fill the full time interval incrementally, and its time complexity is $O(L(|V|\log|V|+|E|))$. Both algorithms require $O(T(|V|+|E|))$ space. T is the average number of turning points in minimum cost functions, and $L > T$ is the average number of subintervals during computation.
- The α-*MORT* approach can return an approximate result faster than the exact algorithms, with the worst error bounded.
- The offline speed profile generation component takes advantages of trajectory, which is cheaper to deploy and has a wider covering range. It consists of a series of processing: map matching, speed data collection, missing value estimation and compression.
- We evaluate the effectiveness and efficiency of our system with extensive experiments on road network and trajectory. The offline component can generate accurate and space-saving speed profiles, and the online component can answer *MORT* problems with both the reduction in the on-road time and the algorithm running time.

This paper extends the work in [13], where we introduced the *MORT* problem, proposed two *MORT* algorithms and tested the performance on synthetic speed profiles. However, the running time of the *MORT* algorithms was long, especially when the trip length and query time interval are both long. Therefore, in this paper, we firstly speed up the query answering by approximating the minimum cost function during route expansion. In fact, our approximation approach can work on all the other time-dependent algorithms. Secondly, we describe how to generate our speed profile from the trajectory, which was not mentioned in [13]. In this way, our system is able to answer queries from real-life rather than only synthetic evaluations.

The rest of the paper is organized as follows. Section 2 discusses the related work. We formally define the minimal on-road time problem in Sect. 3. Section 4 presents the online query answering component with two *MORT* algorithms and analysis. The approximation version α-*MORT* is provided in Sect. 5. Section 6 describes the offline component with the whole process to generate speed profiles using historical trajectories. An empirical study is shown in Sect. 7. Our conclusions can be found in Sect. 8.

2 Related work

In this section, we first review the previous works on modeling time-dependent road network and position our work by discussing the difference from the fastest path problems. Then, we briefly summarize the existing speed profile generation approaches.

2.1 Time-dependent path problems

The simplest model of the time-dependent road network is the discrete time-dependent graph (or *timetable graph*), of which the existence of each edge is time-dependent. A few path planning algorithms such as *earliest arrival time path, latest departure time path, shortest path* and *shortest duration time path* have been proposed on such graphs. Cooke and Halsey [14] proved that these queries could be solved with a modified

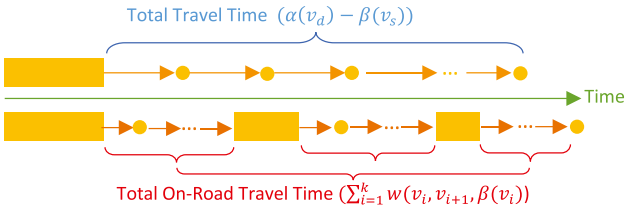

Fig. 3 Comparison between *total travel time* and *on-road travel time*. Thick bar: waiting time on a parking vertex; circle: no waiting on the vertex; arrow: travel time from one vertex to another

version of the *Dijkstra* algorithm. However, it does not scale well with the size of the network. Several techniques are proposed to improve the efficiency [15–17], but they only work on timetable graphs.

A more precise way to describe a time-dependent road network is to use the continuous time-dependent cost function. Fastest path query has been well studied that aims to find a path with the minimum w_{TOT} including waiting time. Dreyfus [10] first showed the time-dependent fastest path problem was solvable in polynomial time if the graph is restricted to have *FIFO* property. Other early theoretical works on this problem include [18] and [19]. However, these algorithms are very difficult to implement, and no empirical evaluation results were reported. Most of the recent path planning algorithms on road network share a common assumption that the travel along a road follows *FIFO* property, which means a vehicle starting earlier will not arrive destination later regardless of the time cost of edges. Due to this property, waiting on a vertex always results in longer total travel time. So these algorithms do not consider waiting on vertices actually. We briefly discuss some representative fastest path algorithms below.

Single Starting-Time Fastest Path (SSFP) algorithm does not allow waiting on the starting vertex. This problem can be solved in $O(|V|\log|V| + |E|)$ time by minor modification on *Dijkstra's algorithm* if *FIFO* property holds [10]. The algorithm can answer both *Earliest Arrival Path* and *Latest Departure Path*, with the same computational complexity.

Interval Starting-Time Fastest Path (ISFP) algorithm allows waiting on the starting vertex in a given starting time interval. However, once departing, no waiting is allowed along the path. The difference between *ISFP* and *MORT* is illustrated in Fig. 3. Moreover, *ISFP* only returns the optimal departure time from starting vertex v_s, while *MORT* needs to determine the optimal departure time from each parking vertex along the path. It is proved in [20] that the theoretical lower-bound of *ISFP* is $\Omega(T(|V|\log|V| + |E|))$ [20], where T is the average number of turning points in the result functions if the weight functions are piecewise linear. Currently, no existing algorithm can achieve this bound because T could be large and it is hard to find the departure time points that would result in the T turning points. Some early works like *DOI* [6] and [9,21] select $k \ll T$ starting time points in the starting time interval and run *SSFP* k times. Obviously, this approach has no guarantee to find the optimal departure time, and both the running time and accuracy highly depend on the choice of k. Kanoulas et al. [4] proposed a path selection and time refinement approach using the heuristic of A^*-algorithm. They computed an arrival time function for each vertex iteratively and used A^*-algorithm to reduce the searching space. However, it is hard to find an appropriate heuristic condition on a time-dependent graph. Ding et al. [5] applied a more precise refinement approach that expanded the time interval step by step rather than computing the entire time interval iteratively. It could avoid unnecessary computations and achieve better performance, although time complexity remained the same. It has a complexity of $O(\alpha(\hat{T})(|V|\log|V|+|E|))$, where $\hat{T}$ is the size of the whole time domain, and $\alpha(\hat{T})$ is the complexity to maintain the time-dependent functions. Although it is not pointed out in their paper, $\alpha(\hat{T})$ actually has a much larger value than the turning point number in the final functions. Other works further extend the static indexes to time-dependent scenario to speed up fastest path query, such as *time-dependent CH* [22] and *time-dependent SHARC* [23]. But they are index level solutions to speed up query answering rather than solving the problem directly.

Although *ISFP* is different from *MORT*, we can adopt it as our baseline algorithm by invoking the algorithms in [4,5] recursively to get an approximate result, as described in Sect. 1. Li et al. [24] and Yang et al. [25] take waiting on intermediate vertices into consideration in their problems. However, they allow waiting on any vertex, which does not make sense in real life. In fact, [24] cannot solve our problem directly and has a time complexity of $O(|V|\log|V|+T|V|+T^2|E|)$, which means it cannot guarantee the optimal result actually since each vertex is visited once. As for [25], they define a time-dependent weight function $w(v_i, v_j, t)$ and a cost function $c(v_i, v_j, t)$ for each edge (v_i, v_j), and aim to find the path with minimum cost, not the minimum weight. But they set the cost functions to linear constants. So rather than confronting with the complex linear piecewise weight functions, they only have to deal with a small set of constant values, which actually simplifies the problem by converting the complex functions to constant values, even though the problem description looks more complicated. Thus, their algorithm cannot find the minimum on-road time (or the minimum weight under their scenario).

From the network point of view, the road network with parking vertices can be treated as a kind of *graph with special nodes*. *Electric vehicle shortest walk problem* [26,27] adopt this model but on static road network. In this problem setting, an electric vehicle has a driving distance limit, and it has to recharge its battery at a power station before the electricity runs out. Given a source vertex and a des-

tination vertex, the problem aims to find the shortest path that the vehicle is able to travel through it. Both [27] and [26] build a sub-network of power station first to solve this problem. It is possible to do this since the network is static and the driving limit is pre-defined. Essentially, it is a special case of *Constraint Shortest Path Problem* [28,29]. If the problem is generalized to be independent on driving distance, the problem becomes *NP-H*. Blokh et al. [30] and Juttner et al. [31] use *Lagrange Relaxation* to find approximate result. Although network model is similar to ours, they do not consider time-dependent cost on edges, which makes it impossible to pre-built a sub-network just as what they do on a static graph. Tong et al. [32,33] also use time-dependent road network and find paths. However, they focus on task assignment based on the existing path algorithms and cannot solve our problem. In fact, their frameworks can utilize our algorithm to provide more functionality.

2.2 Speed profile

Nearly, all the speed profiles using real-world data are either histogram-based [34], or deriving linear functions based on histogram. Demiryurek et al. [11] uses a large amount of sensor data collected in 2 years to build up their speed profile. Since the sensor can work 24 h a day, they are able to collect data in the time slot of 1 min. Then, they organize them into a set of linear functions. Due to the large deployment of the sensor and the long range of collecting time, they do not face the missing value problem. Bakalov et al. [35] describe their system developed in *ESRI*. They store the historical speed as a value between 0 and 1, and derive the actual speed by multiply this value with the *free-flow* speed of that road. Still, since their data are multi-sources rather than only historical trajectory, they are able to derive fine grained speed profile directly without worrying the missing value.

Nevertheless, building a system like the above works is too expensive and not practicable worldwide, so most of the other works in this field focus on constructing a speed profile only from trajectory using histogram method. However, the missing value estimation becomes a big problem. Yang et al. [36] uses *Hidden Markov Models* to estimate the missing values of different time slots. But it is time consuming to train for a large road network and have big issues with the training parameters. Shang et al. [37] applies a Matrix-Factorization-based Collaborative Filtering which we have tested to perform poorly. Xin et al. [38], Asif et al. [39], Shan et al. [40], Widhalm et al. [41], and Guo et al. [42,43] are similar works from machine learning field. They all need loads of trainings on a massive amount of data. However, complicated their formulas are and accurate they claim to be, as we mentioned above, there are actually no real ground truth of a speed profile, so their performances are all based on their own objectives. As long as the error is not too significant and follows *FIFO* property, it is practical to be used in real life.

There are some similar research lines that use trajectory to predict traffic conditions. The first one is called *trajectory regression* [44–46], that aims to find the cost of trajectory based on the existing ones. However, their main purposes are estimating the cost of a given trajectory rather than producing a speed profile on each single road for later route scheduling tasks. The second one is region-based inflow/outflow prediction [47] using deep learning. It can generate the speed profiles on a much coarser granularity, because a region can have hundreds of thousands of roads. Therefore, it cannot help to generate speed profile on each single road.

3 Preliminary

Our problem is twofold: The first one is the minimal on-road time route scheduling, and the second one is related to the speed profile generation from historical trajectories. In this section, we first present the overview of the system framework and then dig into the *MORT* problem definitions.

3.1 Framework overview

Our system has an offline component for speed profile generation and an online component for query answering. Figure 4 gives an overview of it.

In the offline component, historical trajectories are converted into speed profiles by a series of processes. Firstly, trajectories are matched to map such that the speeds on roads are obtained. Then, the speeds are categorized by days and collected by different time slots (for example, 5 min long per slot). Obviously, some slots have no values at all. So we apply missing value estimation methods to fill in the missing speeds. Finally, we compress the raw speed profile to reduce its size while preserving its accuracy.

In the online component, our *MORT* algorithms read the speed profile and answer users' queries. Since the *MORT* algorithm is general in route scheduling, they are able to

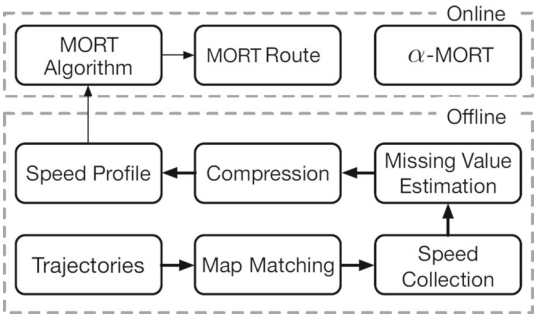

Fig. 4 Framework overview

answer not only the *MORT* queries but also other path planning queries.

3.2 Minimal on-road time route scheduling

A time-dependent road network can be represented as a directed graph $G(V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of ordered pairs of vertices, with a weight function $w : (E, t) \rightarrow \mathbb{R}$ mapping edges to time-dependent real-valued weights. The weight of an edge $e(u, v) \in E$ at time t in a time domain $\mathcal{T}$ is $w(u, v, t)$, which represents the amount of time required to reach v starting from u at time t. In this paper, we only consider the case where the weight of an edge can change over time, but not the case where the structure of a graph can change over time (i.e., V and/or E remain to be static over time). This is a reasonable assumption, as the structure of a road network changes much less frequently compared with the traffic situations. We also define $w(u, v, t) = \infty$ if $(u, v) \notin E$.

A route from u to v in G can be represented as $p =< v_0, v_1, \ldots, v_k >$, where $v_0 = u$, $v_k = v$, and $(v_{i-1}, v_i) \in E$ for any $1 \leq i \leq k$. Let $\alpha(v_i)$ and $\beta(v_i)$ be the arrival and departure time at $v_i \in p$, the *time-dependent* cost of p is the sum of the time-dependent weights of its edges $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i, \beta(v_{i-1}))$. This cost is ∞ by definition if there is no route from u to v in G.

Now let us differentiate two different types of cost for a route: the *total travel time* $w_{\mathrm{TOT}}(p) = \alpha(v_k) - \beta(v_0)$ and the *on-road travel time* $w_{\mathrm{ORT}}(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i, \beta(v_{i-1}))$. Although $w_{\mathrm{ORT}}(p)$ looks identical to $w(p)$ above, the difference here is that for a vertex $v_i \in p$, it is no longer necessary to have $\alpha(v_i) = \beta(v_i)$. In other words, the traveler can stop at a vertex if that can help reduce the on-road travel time. It is trivial to see that $\alpha(v_i) = \beta(v_{i-1}) + w(v_{i-1}, v_i, \beta(v_{i-1}))$ for $i > 0$, and $\beta(v_0)$ is the selected departure time by a path planning algorithm.

The problem to find shortest/fastest path from u to v is to find such a path $p(u, v)$ with minimum cost $w(p)$. Most existing works on this topic have an implicit assumption that for any vertex $v \in p$, $\alpha(v) = \beta(v)$ (e.g., a traveler cannot stop at any vertices along the path). These algorithms focus on w_{TOT} cost. In that case, a traveler who departs earlier will always get to the destination earlier (known as the *FIFO* property [10]). With this setting, travelers always keep $\beta(v) = \alpha(v)$ for any vertex v on a path to achieve optimal w_{TOT}. Some recent works have noticed that, in order to optimize w_{ORT} instead of w_{TOT}, it can be beneficial to delay the departure time at the starting vertex [4,5]. However, there are more vertices than just the source vertex in a road network where a vehicle can stop for a period of time. Let $V' \subseteq V$ be a set of *parking vertices* in G where a vehicle can wait *voluntarily* for a minimum amount of time t_{min} before traveling again. In other words, $\beta(v) - \alpha(v) \geq v.t_{min}$ if $v \in V'$, and $\beta(v) = \alpha(v)$ if $v \in V - V'$. The minimum waiting time $v.t_{min}$ is aimed to avoid useless short waiting time or meet user's minimum staying time requirement by providing a user pre-defined lower bound. If they are all set to 0, then the waiting time could be arbitrary on each waiting vertices. This should not be confused with the case that a vehicle stops in a traffic jam or in front of a traffic light; these forced stops are captured by the weight function of $w(u, v, t)$ already.

We are ready to define the problem we address in this paper as follows.

Definition 1 (*Minimal On-Road Time Route Scheduling Problem*) Given a directed graph $G = (V, E)$ with a set of parking vertices $V' \subseteq V$, each of which has a minimum staying time $v_i.t_{min}$ and a time-dependent edge weight function w, a query $Q_{MORT}(v_s, v_d, t_{s1}, t_{s2}, t_d)$ is to find a route from v_s to v_d, represented as $p =< v_0, v_1, \ldots v_k >$, such that: (1) $v_s = v_0$ and $v_d = v_k$; (2) $\beta(v_i) = \alpha(v_i)$ if $v_i \in V - V'$ and $\beta(v_i) - \alpha(v_i) \geq v_i.t_{min}$ if $v_i \in V'$; (3) $t_{s1} \leq \beta(v_s) \leq t_{s2}$; (4) $\alpha(v_d) \leq t_d$; and (5) $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i, \beta(v_{i-1}))$ is minimal among all possible routes meeting the previous conditions.

Condition (1) means that p is a route from v_s to v_d, and condition (2) allows the traveler to stop and wait only at a parking node for a minimum period of time. Conditions (3) and (4) define that the traveler must depart v_s during the specified time interval and must arrive at v_d before the given latest arrival time t_d. If there does not exist a route meeting these four conditions, the cost to travel from v_s to v_d is defined as ∞. Condition 5 requires the route to have the minimal on-road travel time.

If the edge weight is not time dependent (i.e., the weight for each edge is static), a *MORT* query reduces to traditional shortest path queries in a static road network [1]. Besides, the time-dependent query studied in [4,5] is a special case of the *MORT* query where parking node set $V' = \{v_s\}$.

3.3 Speed profile generation from trajectory

A trajectory $tr_i =< (x_1^i, y_1^i, t_1^i), \ldots, (x_m^i, y_m^i, t_m^i) >$ is series of GPS points, where each point (x_1^i, y_1^i, t_1^i) contains a longitude x, a latitude y and a timestamp t. Given a set of trajectories $Tr = (tr_1, tr_2, \ldots, tr_n)$ and a directed graph $G = (V, E)$, speed profile generation is to derive the time-dependent edge weight function $w(u, v, t)$ for each $(u, v) \in E$ using Tr. Such a process involves map-matching, speed collection, missing value estimation and compression.

The first step is converting the trajectories to road speeds using map-matching. For a trajectory $tr_i =< (x_1^i, y_1^i, t_1^i), \ldots, (x_m^i, y_m^i, t_m^i) >$, we can derive a consecutive series of edges $E_i =< e_1, \ldots, e_m >$ where $e_j e_{j+1} \neq \phi$, with each point (x_j^i, y_j^i, t_j^i) is attached to some edge $e_k \in E_i$. Therefore, with the information of the road network distance,

the sequence of roads and time difference between any two consecutive points, it is trivial to get the speeds of their corresponding roads at different times during a day.

The second step is categorizing the speed data and organizing them into time slots. We start with categorizing the speed data by weekdays and weekends, because they follow different traffic patterns. After that, we collect them in different time slots. If we set the slot size to 5 min, then we have 288 time slots of a day. Or if we set the size to half an hour, then we have 48 slots. With the data collected in each slot, we prune out the outlier data and compute the average speed of the remaining as the speed for this time slot.

The third step is missing value estimation. Obviously, quite a number of time slots of many roads may not have speed data at all, especially when the slot size is small. For example, in the 5 min one, 85% of the slots have no data. In order to cope with it, we use several approaches to estimate the missing values.

The last step is compression. The speed profile generated by the previous steps is histogram-based and has a fixed size of $K \times |E|$, where K is the slot size and $|E|$ is number edges in G. However, many of the speed values are close or even equal to each other, or follow some function distribution. Such a speed profile is space consuming. To reduce the size while preserving the accuracy, we use *linear piecewise aggregation* to convert the histogram data to linear function, because the speed data can be viewed as time series data. In this way, we can get an weight function $w(u, v, t)$ for each edge (u, v).

For the ease of exposition, we first explain our query algorithm in Sect. 4 and approximation approaches in Sect. 5, assuming that the linear piecewise function is given. Then, in Sect. 6, we will elaborate on how the linear piecewise function is derived.

4 MORT algorithms

In this section, we describe our *MORT* algorithms in detail. The key idea is that we define and maintain a variational piecewise *Minimum Cost Function* $C_i(t)$ for each vertex v_i. $C_i(t)$ returns different minimal on-road travel time from v_s to v_i given different arrival time t, so it has the potential to model traffic tendency more accurately. Based on the new cost function, we design two algorithms to expand the *MORT* route step by step in a *Dijkstra* way: (1) the *Basic MORT algorithm* constructs $C_d(t)$ by updating $C_i(t)$ of each visited vertex over the whole time interval, and finishes expanding until $C_d(t)$ is stable; (2) the *Incremental MORT algorithm* decomposes $C_d(t)$ into different parts according to the query time subintervals, and finishes expanding until each part of $C_d(t)$ is complete. Both of these algorithms do not require the graph to follow *FIFO* property. Although our route expanding algorithms are able to find the *MORT* time, its result does not contain a schedule, which is the expected output of *MORT* problem. To address that, *route retrieval* is introduced to generate the final results. Considering scalability is important for route scheduling, we present the correctness and complexity analysis at the end of each subsection.

4.1 Algorithm outline

Given a time-dependent graph $G(V, E)$ and a *MORT* query $Q_{MORT}(v_s, v_d, t_{s1}, t_{s2}, t_d)$, the proposed algorithm generates the minimal on-road time $R_{p^*_{s,d}}$ and the corresponding route with traveling schedule $p^*_{s,d}$. The whole process can be divided into three parts as below:

1. *Active Time Interval Profiling (ATI)* computes the active time interval T_i for each vertex v_i, which is bounded by a pair of earliest arrival time $v_i.t_{EA}$ and latest departure time $v_i.t_{LD}$.
2. *Route Expansion* finds the route with minimum on-road travel time in a *Dijkstra* way and produces the *Minimum Cost Functions* of the visited vertices.
3. *Route Retrieval* returns the actual route schedule with user-specified arrival time.

In the following subsections, we will introduce each part of the proposed algorithm thoroughly except for the route expansion part. The full details of the route expansion which are the major contributions in this work will be presented in Sects. 4.2 and 4.3, respectively. We further explain how to apply our algorithms to different scenarios in Sect. 4.4.

4.1.1 Active time interval computation (ATI)

The MORT query specifies a departure interval $[t_{s1}, t_{s2}]$ on v_s and a latest arrival time t_d on v_d. With these constraints, the route schedule is roughly outlined but loose for other vertices. If the graph does not follow *FIFO*, we have to use this loose time interval. Otherwise, we could reduce the computation load by computing an *active time interval (ATI)* for each vertex in the proposed algorithms. An *active time interval (ATI)* of a vertex v_i is denoted as $T_i = [v_i.t_{EA}, v_i.t_{LD}]$, which is bounded by a earliest arrival time $v_i.t_{EA}$ (we cannot arrive v_i any earlier) and a latest departure time $v_i.t_{LD}$ (we will never arrive v_d before t_d if it departs from v_i any later). It models a vehicle's possible occurrence interval on the corresponding vertex under the query constraints (t_s and t_d). *ATI* is very important for the proposed algorithm since it is the basis of the other parts. In the following, we will introduce how the *ATI* is computed for each vertex.

ATI, as well as all the following calculations, is computed from speed profile. In a speed profile, each edge (v_i, v_j) is associated with a function $w(v_i, v_j, t)$ whose parameter

is t and output is time cost. Compared to [25], function $w(v_i, v_j, t)$ is a combination of consecutive linear functions rather than constant values. It obeys the *FIFO* and serves in the *route expansion*. Notice that when t is given, we use $w(v_i, v_j, t)$ to represent the time cost of traveling from v_i to v_j at time t. The speed profile is then instantiated as $\{(t_0, w(v_i, v_j, t_0)), \ldots, (t_k, w(v_i, v_j, t_k))\}$, and the intermediate values between points are computed linearly. Figure 1b–f illustrates an example of speed profile.

Given the proposed speed profile, the earliest arrival time of each vertex is computed by performing *SSFP* from v_s at t_{s1}. As for the latest departure time, we have to compute from v_d at t_d reversely, both in time and in vertex order. After two rounds of *SSFP*, each vertex obtains its active time interval, and all the future computations will be based on the active time intervals. The *ATI* has the same time complexity as *Dijkstra*, which is $O(|V| \log |V| + |E|)$.

We use the road network in Fig. 1 and query $Q_{MORT}(v_1, v_5, 0, 30, 130)$ as an example. $ATI(v_1, v_5, 0, 30, 130)$ generates the following active time intervals: $T_1 = [0, 25]$, $T_2 = [40, 65]$, $T_3 = [70, 95]$, $T_4 = [95, 125]$ and $T_5 = [105, 130]$.

4.1.2 Minimum cost function

In order to model the correlations between time and cost, we construct a *minimum cost function* whose value varies with arrival time for each vertex, instead of defining the minimum cost which is constant over time in [25]. Accordingly, the output of route expansion in our work is the minimal of v_d's minimum cost function. Since the minimum cost function is the basis of the two proposed route expansion algorithms, we present the definition and construction of the minimum cost function in this part.

The minimum cost function, denoted as $C_i(t)$, monitors the minimum on-road cost of traveling from v_s to v_i that arrives on time t. The minimum value of $C_i(t)$ is equivalent to the minimum on-road time (*MORT*) from v_s to v_i . For example, $C_i(300) = 50$ means when it starts traveling from v_s at t_s and arrives on v_i at time 300, the minimum on-road travel time (*MORT*) is 50. Accordingly, for the destination vertex v_d, the *MORT* is $min(C_d(t))$. In addition, for a parking vertex v_i^p, the value of dependent variable of $C_i^p(t)$ has a *non-increasing* property:

Lemma 1 $\forall v_i \in V'$ *and* $\forall v_i.t_{EA} \leq t_a < t_b \leq v_i.t_{LD}$, $C_i^p(t_a) \geq C_i^p(t_b)$

The non-increasing property reveals a natural fact: If one route schedule arriving at t_b takes higher cost than another arriving at t_a, we should choose the latter one and wait from t_a to t_b, which reduces the on-road time from $C_i^p(t_b)$ to $C_i^p(t_a)$. The non-increasing property indicates that waiting is necessary to decrease the on-road travel time (Table 1).

Table 1 Important notations

Notation	Description
$v.t_{min}$	Minimum waiting time on v
T_i	Active time interval of v_i
I_i	$[v_i.t_{EA}, \tau_i] \subseteq T_i$
τ_i	Upper bound of I_i
$C_i(t)$	Minimum cost function of v_i
$g_{f,i}(t)$	$C_f(t) + w(v_f, v_i, t)$
$g'_{f,i}(t)$	Non-increasing version of $g_{f,i}(t)$
$C'_i(t)$	$min(C_i(t), g_{f,i}(t))$
$C^*_d(t)$	Optimal result on destination $min(C_d(t))$
$A_i(t)$	Approximate minimum cost function
$\hat{E}$	Edges along a route
$\lvert\hat{E}\rvert$	Number of edges along $\hat{E}$
$\lvert\lvert\hat{E}\rvert\rvert$	Length of $\hat{E}$
α	Pruning budget such that $A^*_d(t) \geq \alpha C^*_d(t)$
$SP_{i,j}$	Speed of edge e_i at time slot t_j
SP_i	Speed vector of edge e_i

$C_i(t)$ is linear piecewise because it is constructed from the speed profile which is also linear piecewise. Thus, a minimum cost function $C_i(t)$ equals a set of consecutive discrete linear functions. These functions share the end points and are maintained in the ascending order of time. Based on that, the cost function of a vertex is denoted as an ordered point set $S_i = \{(t_0, C_i(t_0)), \ldots, (t_k, C_i(t_k))\}$. The update of S_i is achieved by merge. For instance, suppose $C'_i(t)$ is the current minimum cost function of v_i, and $C''_i(t)$ is another minimum cost function provided by another route to v_i, the new $C_i(t)$ is formed by merging the smaller parts of these two functions: $min(C'_i(t), C''_i(t))$.

4.1.3 Route retrieval

The route retrieval generates the route schedule based on the user-specified arrival time using the minimum cost functions. For each turning point in the ordinary vertices' minimum cost functions, we store its predecessors. For the parking vertices, apart from the predecessors for the turning points, we also need to store the points that happen to have the same value as the current cost (no turning point added because it is not smaller). This predecessor cache has the same space complexity as the minimum cost functions.

Suppose t is a user-specified arrival time. We can traverse the vertices back from v_d at time t. Suppose we are visiting v_i at t_i. Firstly, if v_i is an ordinary vertex, we find the latest turning point $(t'_i, C_i(t'_i))$ in $C_i(t)$ such that $t'_i \leq t_i$, and use its predecessor as the next visiting vertex. The arrival time is the same as t_i. Secondly, if v_i is a parking vertex, we also find the latest turning point $(t'_i, C_i(t'_i))$ in $C_i(t)$ with $t'_i \leq t_i$.

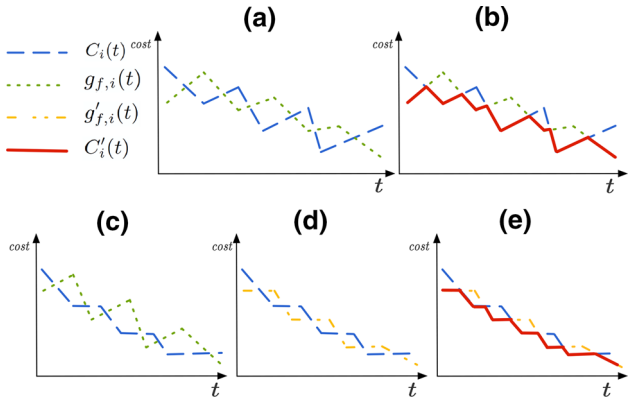

Fig. 5 Minimum cost function update. **a** $g_{f,i}(t)$ and $C_i(t)$ for ordinary vertex v_i. **b** Result of $min(g_{f,i}(t), C_i(t))$ for ordinary vertex v_i. **c** $g_{f,i}(t)$ and $C_i(t)$ for parking vertex v_i, $C_i(t)$ is non-increasing. **d** $g_{f,i}(t)$ applies non-increasing. **e** Result of $min(g_{f,i}(t), C_i(t))$ for parking vertex v_i

However, the arrival time is t'_i rather than t_i. If the turning point has more than one predecessor, or the parking vertex has more than one points with the same cost, we can traverse the graph in a *DFS* way to output more than one routes for users to choose. Obviously, this approach takes $O(k)$ time, where k is the number of vertices along the route.

4.2 Basic MORT algorithm

The *Route Expansion* in *Basic MORT* algorithm uses a *Dijkstra* way to find the *MORT* from v_s to other vertices. Instead of using the *shortest distance* as the sorting key, we use the minimum value of each vertex's $min(C_i(t))$. Each time we visit a vertex, we update its neighbors' $C_i(t)$ over their *ATI*, until $C_d(t)$ is guaranteed stable. We first describe how to update the minimum cost function $C_i(t)$ in Sect. 4.2.1 and then present route expansion in Sect. 4.2.2. The correctness and complexity analysis are provided in Sects. 4.2.3 and 4.2.4.

4.2.1 Minimum cost function update (MCFU)

Each time we visit a vertex, we update its out-neighbor's $C_i(t)$. From v_i's point of view, its $C_i(t)$ can only be updated by its in-neighbors. Suppose v_f is v_i's in-neighbor, $C_f(t)$ is v_f's minimum cost function and $w(v_f, v_i, t)$ is the weight function on edge (v_f, v_i). We use $g_{f,i}(t') = C_f(t) + w(v_f, v_i, t), t' = t + w(v_f, v_i, t)$ to denote the cost to travel from v_s to v_i via v_f. Depending on whether v_i is a parking vertex or not, we update $C_i(t)$ differently.

The update of ordinary $C_i(t)$ has two steps as shown in Fig. 5a, b. We first calculate $g_{f,i}(t)$(dot line). Then, we compare $g_{f,i}(t)$ with original $C_i(t)$ (dash line) and use the smaller parts of the two functions as the new minimum cost function $C'_i(t)$ (solid line). We use the line segment intersection detection technique to compute $C'_i(t) = min(C_i(t), g_{f,i}(t))$.

However, if v_i is a parking vertex, we cannot use $g_{f,i}(t)$ directly since the result of $min(C_i(t), g_{f,i}(t))$ may not follow non-increasing property. So we convert $g_{f,i}(t)$ to its non-increasing version $g'_{f,i}(t)$ first before computing $C'_i(t)$. Figure 5c shows the non-increasing $C_i(t)$ and a ordinary $g_{f,i}(t)$. We convert $g_{f,i}(t)$ into its non-increasing version $g'_{f,i}(t)$ in Fig. 5d, and then compute $C'_i(t)$ in Fig. 5e. The correctness is guaranteed by the following lemma.

Lemma 2 *If both $C_i(t)$ and $g'_{f,i}(t)$ are non-increasing, then $C'_i(t) = min(C_i(t), g'_{f,i}(t))$ is also non-increasing.*

Proof $\forall t_a < t_b \Rightarrow C_i(t_a) \geq C_i(t_b), g_{f,i}(t_a) \geq g_{f,i}(t_b)$. (1) If $min(C_i(t_a), g_{f,i}(t_a)) = C_i(t_a)$ and $min(C_i(t_b), g_{f,i}(t_b) = C_i(t_b), C_i(t_a) \geq C_i(t_b)$, non-increasing holds. (2) If $min(C_i(t_a), g_{f,i}(t_a)) = g_{f,i}(t_a)$ and $min(C_i(t_b), g_{f,i}(t_b) = C_i(t_b), g_{f,i}(t_a) \neg g_{f,i}(t_b) \neg C_i(t_b)$, non-increasing holds. The remaining two situations are similar. □

In order to guarantee the minimum staying time on the parking vertices, we attach a user-specified value $v_i.t_{min}$ on each $v_i \in V'$. When computing $g_{f,i}(t)$ from a parking vertex v_f to v_i, the departure time from v_f is changed to $t' = t + v_f.t_{min}$. Thus, the arrival time on v_i further grows to $t'' = t' + w(v_f, v_i, t')$. So $g_{f,i}(t'') \leftarrow C_f(t') + w(v_f, v_i, t')$.

The details of *MCFU* are shown in Algorithm 1. Suppose v_f is the current visiting vertex and v_i is v_f's out-neighbor. *MCFU* computes the updated $C'_i(t)$ using $C_f(t)$ and the edge weight $w(v_f, v_i, t)$. It works in a *sweeping-line* way. Lines 2–6 compute the cost to v_i via v_f. If v_f is a parking vertex, then minimum staying time is applied. If v_i is a parking vertex, a non-increasing version $g'_{f,i}(t)$ is generated (Lines 7–8). Then, it visits the line segments in the $C_i(t)$ and $g'_{f,i}(t)$ together one by one. Initially, it retrieves the first line segments in $C_i(t)$ and $g'_{f,i}(t)$ (Lines 9–10), and their corresponding end points (p_1, p_2) and (p'_1, p'_2) (Lines 12–13). Lines 14–17 use the line segment intersection technique, which tells the position relation of two lines by computing d_1, d_2, d_3 and d_4, as illustrated in Fig. 6. If $d_1 > 0, d_2 < 0, d_3 < 0$ and $d_4 > 0$ (Line 18), it is guaranteed that the line segments has an intersection point p' and line segment (p_1, p') should appear in $C'_i(t)$. If $d_1 < 0, d_2 > 0, d_3 > 0$ and $d_4 < 0$ (Line 22), the line segment (p'_1, p') should appear in $C'_i(t)$. Then, the corresponding points are updated in Line 21 or Line 25. The loop recurs until it reaches the last end points. Suppose the active time interval has T time units. In the worst case, there are T end points in the cost function. Within the update of each line segment, it only costs constant time. So the time complexity of the Algorithm 1 is $O(T)$.

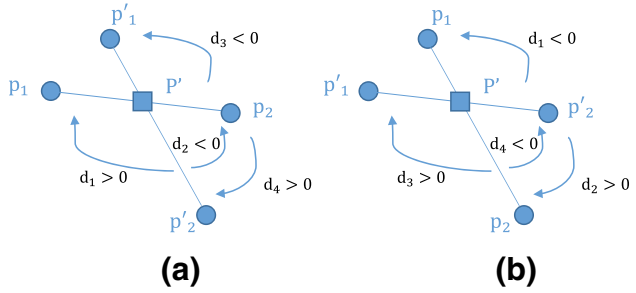

Fig. 6 Line segment intersection

Algorithm 1: Minimum Cost Function Update (MCFU)

Input: v_i's minimum cost function $C_i(t)$, v_f's minimum cost function $C_f(t)$, the cost function from v_f to v_i: $w(v_f, v_i, t)$ and minimum staying time $v_f.t_{min}$ on v_f
Output: v_i's new minimum cost function $C'_i(t)$

1 **begin**
2 **if** $v_f \in V'$ **then**
3 $g_{f,i}(t'') \leftarrow C_f(t') + w(v_f, v_i, t')$
4 $t' \leftarrow t + v_f.t_{min}, t'' \leftarrow t' + w(v_f, v_i, t')$
5 **else**
6 $g_{f,i}(t') \leftarrow C_f(t) + w(v_f, v_i, t), t' \leftarrow t + w(v_f, v_i, t)$
7 **if** $v_i \in V'$ **then**
8 $g'_{f,i}(t) \leftarrow Non - Increase((g_{f,i}(t))$
9 $t_1 \leftarrow S_i[0], t'_1 \leftarrow S_i[1]$ //S_i: time points in $C_i(t)$
10 $t_2 \leftarrow S_f[0], t'_2 \leftarrow S_j[1]$ //S_f: time points in $g'_{f,i}(t)$
11 **while** $t_1 \neq S_i.end$ *and* $t_2 \neq S_j.end$ **do**
12 $p_1 \leftarrow (t_1, C_i(t_1)), p_2 \leftarrow (t_2, C_i(t_2))$
13 $p'_1 \leftarrow (t'_1, g'_{f,i}(t'_1)), p'_2 \leftarrow (t'_2, g'_{f,i}(t'_2))$
14 $d_1 \leftarrow Direction(p'_1, p'_2, p_1)$
15 $d_2 \leftarrow Direction(p'_1, p'_2, p_2)$
16 $d_3 \leftarrow Direction(p_1, p_2, p'_1)$
17 $d_4 \leftarrow Direction(p_1, p_2, p'_2)$
18 **if** $d_1 > 0$ *and* $d_2 < 0$ *and* $d_3 < 0$ *and* $d_4 > 0$ **then**
19 $(t', C_i(t')) \leftarrow$ intersection point
20 $C'_i(t).insert(t', C_i(t'))$
21 $t_1 \leftarrow t', t'_1 \leftarrow t'_2, t'_2 \leftarrow S_j.next$
22 **else if** $d_1 < 0$ *and* $d_2 > 0$ *and* $d_3 > 0$ *and* $d_4 < 0$ **then**
23 $(t', C_i(t')) \leftarrow$ intersection point
24 $C'_i(t).insert(t', C_i(t'))$
25 $t'_1 \leftarrow t', t_1 \leftarrow t_2, t_2 \leftarrow S_i.next$
26 **return** $C'_i(t)$
27 **Function** *Direction*(p_i, p_j, p_k)
28 **return** $(p_k - p_i) \times (p_j - p_i)$

4.2.2 Basic route expansion algorithm

Route expansion algorithm maintains a priority queue Q that uses $min(C_i(t))$ as keys to store all the vertices. Each time we pop out the top vertex and update its out-neighbors' $C_i(t)$. This procedure runs on until $C_d(t)$ is guaranteed stable. The details are described in Algorithm 2. Lines 2–5 initialize the minimum cost function of each vertex by adding the two end points $(v_i.t_{EA}, v_i.t_{EA} - t_{s1})$ and $(v_i.t_{LD}, \infty)$. Obviously, the source vertex's cost is always 0. Then, these minimum cost functions are organized into a priority queue Q ordered by their $min(C_i(t))$. Each time we pop up the vertex v_i with the smallest $min(C_i(t))$ value in Q and use it to update the minimum cost functions of its out-neighbors v_j using algorithm 1 (Line 12). If $C_j(t)$ has changed and v_j is out of Q, we insert the new function back to Q. If it is changed but still in Q, we just update its key (Lines 13–17). The algorithm terminates either when Q becomes empty (Line 7) or when the top function's smallest value is larger than v_d's minimum on-road cost (Lines 9–10).

Algorithm 2: Route Expansion Algorithm

Input: $G(V, E)$, $Q_{MORT}(v_s, v_d, t_{s1}, t_{s2}, t_d)$
Output: $R_{p^*_{s,d}}$

1 **begin**
2 **for** $v_i \in V$ **do**
3 $C_i(v_i.t_{EA}) \leftarrow v_i.t_{EA} - t_{s_1}$
4 $C_i(v_i.t_{LD}) \leftarrow \infty$
5 Let Q be a priority queue initially containing pairs $(min(C_i t), v_i)$, ordered by $min(C_i t)$ in ascending order
6 $Q.insert(min(C_s(t)), v_s)$
7 **while** Q *is not empty* **do**
8 $v_i \leftarrow Q.pop()$
9 **if** $min(C_i(t)) \geq min(C_d(t))$ **then**
10 *break*
11 **for** $v_j \in v_i$*'s out-neighbors* **do**
12 $C'_j(t) = MCFU(C_j(t), C_i(t), w(v_i, v_j, t))$
13 **if** $C'_j(t) \neq C_j(t)$ **then**
14 **if** $v_j \in Q$ **then**
15 $Q.Update(min(C_j(t)), v_j)$
16 **else**
17 $Q.insert(min(C_j(t)), v_j)$
18 **return** $min(C_d(t))$

4.2.3 Correctness

Theorem 1 *Algorithm 2 finds the MORT.*

Proof Initially, the top of Q is $min(C_s(t))$, which is 0 because v_s is the starting vertex. Then, its out-neighbors can all get their *MORT* after updated from v_s. Suppose v_i is the current top item of Q and v_j is v_i's out-neighbor. If $min(C_j(t)) < min(C_i(t))$, then $\forall \Delta > 0, min(C_i(t)) + \Delta > min(C_j(t))$. So v_i cannot update $C_j(t)$'s minimum value. In fact, v_j has already found its *MORT* that no vertex in Q can reduce it. But the other parts of $C_j(t)$ could be changed. So if $C_j(t)$ is changed, it is inserted back to Q. If $min(C_i(t)) < min(C_j(t))$, v_j might find a better route via v_i and gets updated. And since $min(C_i(t)) < min(C_k(t)), \forall v_k \in Q$, it is ensured that $min(C_i(t)) < min(C_j(t)) + \Delta, \forall \Delta > 0$. Thus, v_i has found its *MORT* that no vertex in Q can reduce it. Finally, after the $min(C_i(t)) > min(C_d(t))$ pops out from

Q, it is guaranteed that no vertex in Q can update $min(C_d(t))$. Thus, v_d has found its *MORT*. □

4.2.4 Complexity analysis

As mentioned previously, the time complexity of the *ATI* algorithm is $O(|V|\log|V|+|E|)$. As for the *Route Expansion* algorithm, we use *Fibonacci Heap* [48] to implement the priority queue. T is used to denote the average number of turning points in $C_i(t)$, which indicates the average number of times a vertex's minimum cost function would be updated among all the vertices. So on average, $C_i(t)$ could be updated T times, which means v_i is visited T times. The maximum number of elements in Q is $|V|$, and it takes $\log|V|$ time to pop out the top element. So it takes $O(T|V|\log|V|)$ time in total to retrieve the top elements in Q. Each edge might be visited T times to update the corresponding minimum cost function, and *MCFU* also takes $O(T)$ time. So the update part of the algorithm takes $O(T^2|E|)$ time. Thus, the total time complexity of *Basic MORT Algorithm* is $O(T|V|\log|V|+T^2|E|)$.

As for the space complexity, the speed profile takes $O(T|E|)$ space, the minimum cost function takes $O(T|V|)$ space, and the graph itself takes $O(|V|+|E|)$ space. Hence, the total space complexity is $O(T(|V|+|E|))$.

4.3 Incremental MORT algorithm

Unlike *Basic MORT* which updates the minimum cost function on the whole active time interval repeatedly, *Incremental MORT Algorithm* uses *Incremental Route Expansion* to build the minimum cost function for each vertex v_i in its $T_i = [v_i.t_{EA}, v_i.t_{LD}]$ subinterval by subinterval incrementally, which could reduce unnecessary computations.

4.3.1 Incremental route expansion algorithm

Suppose for a subinterval $I_i = [v_i.t_{EA}, \tau_i] \subseteq T_i = [v_i.t_{EA}, v_i.t_{LD}]$, we have already computed its minimum cost function $C_i(I_i)$. Then, we extend I_i to a larger subinterval $I_i' = [v_i.t_{EA}, \tau_i'] \subseteq T_i$ where $\tau_i' > \tau_i$ and make sure $C_i(I')$ is refined. It should be noted that the current $C_i(t)$ is constructed by v_i's in-neighbors, and refinement means specifying a larger subinterval within which the minimum cost function is stable. After that, we update v_i's out-neighbor v_j's $C_j(t)$ in its corresponding time interval $[\tau_j^1, \tau_j^2]$. $v_j's$ $C_j(t)$ will be refined when we visit them. When τ_i reaches $v_i.t_{LD}$, $C_i(t)$ is guaranteed to be refined over T_i. When τ_d reaches t_d, the algorithm terminates. The details are shown in Algorithm 3. It is made up of two main parts: *Arrival Time Interval Extension* to determine the next subinterval to refine, and *Minimum Cost Function Update*.

Algorithm 3: Incremental Route Expansion Algorithm

Input: $G(V,E)$, $Q_{MORT}(v_s, v_d, t_{s1}, t_{s2}, t_d)$
Output: $R_{p^*_{s,d}}$

```
1  begin
2      C_s(t_s) ← 0, C_s(v_s.t_LD) ← 0, τ_s ← t_s
3      for v_i ∈ V/{v_s} do
4          C_i(v_i.t_EA) = v_i.t_EA − t_s, τ_i ← v_i.t_EA
5      Let Q be a priority queue initially containing pairs (τ_i, C_i(t)),
       ordered by τ_i in ascending order
6      while |Q| ≥ 2 do
7          (τ_i, C_i(t)) ← Q.pop()
8          (τ_k, C_k(t)) ← Q.top()
9          τ_i' ← τ_k + min{w(v_f, v_i, τ_k)|v_f is v_i's in-neighbor}
10         for v_j is v_i's out-neighbor do
11             if v_i ∈ v' then
12                 C_j'(t'') ← C_i(t') + w(v_i, v_j, t')
13                 t' ← t + w(v_i, v_j, t), t'' ← t' + v_i.t_min
14             else
15                 C_j'(t') ← C_i(t) + w(v_i, v_j, t)
16                 t' ← t + w(v_i, v_j, t)
17             t ∈ [τ_i, τ_i']
18             if v_j ∈ V' then
19                 C_j'(t) ← Non − Increase(C_j'(t))
20             τ_j^1 = τ_i + w(v_i, v_j, τ_i)
21             τ_j^2 = τ_i' + w(v_i, v_j, τ_i')
22             C_j(t) ← min(C_j(t), C_j'(t), t ∈ [τ_j^1, τ_j^2]
23             Q.update(τ_j, C_j(t))
24         τ_i ← τ_i'
25         if v_i = v_d and τ_i ≥ t_d then
26             return min(C_i(t))
27         else if τ_i < v_i.t_LD then
28             Q.insert((τ_i, C_i(t)))
29     R_{p^*_{s,d}} = min(C_d(t))
```

Initially, we set v_s's cost function to 0 in its active time interval and set τ_s to the query's starting time (Line 2). Then, we set the other vertices' cost functions to their earliest arrival time minus t_s and the corresponding τ_i to their earliest arrival time $v_i.t_{EA}$ (Lines 3–4). At this stage, the subintervals of the vertices are empty. So, all cost functions are refined. We use a priority queue Q to organize the information. The elements we insert into Q are pairs of $(\tau_i, C_i(t))$ ordered by τ_i. The while loop (Lines 6–28) updates the minimum cost functions and refines the subintervals. For each element in Q, it is ensured that its minimum cost function is well refined in its subinterval $[v_i.t_{EA}, \tau_i]$.

Arrival Time Interval Extension (Lines 7–9): Each time we pop out the top pair $(\tau_i, C_i(t))$ from Q. As defined, $C_i(t)$ is well refined within subinterval $[v_i.t_{EA}, \tau_i]$. Then, we need to expand this subinterval to a later arrival time such that its well-refined claim still holds. Recall that the elements in Q are sorted by τ which is the arrival time of each vertex. It is obvious that τ_i is no bigger than any τ in Q, and the current top pair $(\tau_k, C_k(t))$ has the smallest τ in Q. Thus, for any v_i's in-neighbor v_f, its refined time interval's upper bound $\tau_f \geq \tau_k$. If $C_i(t)$ needs to be updated by v_f, it would be later than

$\tau_f + w(v_f, v_i, \tau_k)$. Suppose v_f has the smallest travel cost at τ_k among all v_i's in-neighbors, then no vertex can change $C_i(t)$ before $\tau_k + w(v_f, v_i, \tau_k)$). That is to say, $C_i(t)$ is well refined in subinterval $[\tau_i, \tau_i']$, where $\tau_i' = \tau_k + w(v_f, v_i, \tau_k)$ (Line 9).

Minimum Cost Function Update (Lines 10–23): For each out-neighbor v_j of v_i, we compute its $C_j(t)$ that departs from v_i within $[\tau_i, \tau_i']$. This part is similar to *Basic MORT*'s but it works on a smaller time interval. If v_i is a parking vertex, we apply minimum staying time on it (Line 11–13). If its neighbor v_j is a parking vertex, we apply the non-increasing property on it. Then, we compute the corresponding new subinterval: lower bound τ_j^1 is $\tau_i + w(v_i, v_j, \tau_i)$ and upper bound τ_j^2 is $\tau_i' + w(v_i, v_j, \tau_i')$. Finally, we compare the new $C_j'(t)$ with the existing $C_j(t)$ and use the smaller one as the newly computed $C_j(t)$, and update v_j's function in Q. It should be noted that although we have updated $C_j(t)$ in a new subinterval, it is still not well refined within it. It is only when we actually visit v_j as the top element in Q that its refined subinterval can be expanded.

After updating, we go back to see v_i itself. We first set τ_i to its new value τ_i' (Line 24). If τ_i has already reached its latest departure time, then $C_i(t)$ is fully refined and we will not need it anymore. Otherwise, it is still not well refined and thus we insert it back to Q with the new τ_i as the sorting key (Line 28). If v_d is fully refined within its active time interval, the algorithm terminates. As for the minimum value of $C_d(t)$, it is trivial to maintain.

4.3.2 Running example

We continue with the example used in Sect. 4.1.1. After running $ATI(v_1, v_5, 0, 30, 130)$, we can get the corresponding initial τ values (earliest arrival times): $\tau_1 = 0, \tau_2 = 40, \tau_3 = 70, \tau_4 = 95$ and $\tau_5 = 105$. Thus, the initial elements in Q are $< (\tau_1 = 0, C_1(t)), (\tau_2 = 40, C_2(t)), (\tau_3 = 70, C_3(t)), (\tau_4 = 95, C_4(t)), (\tau_5 = 105, C_5(t)) >$. $C_0(t)$ has two points $(0, 0)$ and $(25, 0)$, and the other $C_i(t)$ only has one point (τ_i, τ_i).

In the first iteration, v_1 has the smallest τ in Q, so we pop v_1 out of Q. The current top element in Q is $(\tau_2 = 40, C_2(t))$, which has the earliest refined arrival time in Q. Thus, we use $\tau_2 = 40$ as the base time. v_1 has no in-neighbor, so $min(w(v_f, v_1, 40)) = \infty > v_1.t_{LD}$. Then, v_1 is well refined in its active time interval. Now, we update v_1's out-neighbors in the refined time interval [0,25]. Because v_2 is v_1's only out-neighbor and the edge cost function of (v_1, v_2) is $w(v_1, v_2, t)$, we compute $C_2(t)$ on time interval $[0 + w(v_1, v_2, 0), 25 + w(v_1, v_2, 25)] = [40, 65]$. It should be noted that although $C_2(t)$ is newly computed, τ_2 remains 40, which means the $C_2(t)$ from $t = 40$ is still unrefined and might be changed by other vertices.

In the second iteration, the current Q is $< (\tau_2 = 40, C_2(t)), (\tau_3 = 70, C_3(t)), (\tau_4 = 95, C_4(t)), (\tau_5 = 105, C_5(t)) >$. We pop out the top element v_2 and visit it. The current top element is $\tau_3 = 70$, so none of the in-queue vertices' refined latest arrival time is earlier than 70, which means all the vertices's time interval before 70 has been used to update their out-neighbors. For v_2's in-neighbor v_1, if it departs at $t = 70$, it will arrive v_2 at $70 + w(v_1, v_2, 70) = 97.5$. So it is guaranteed that no vertices can change $C_2(t)$ in time interval [40, 97.5]. Thus, $C_2(t)$ is refined in [40, 97.5], and its new τ_2 is extended to 97.5. However, since $97.5 > v_2.t_{LD}$, v_2 is also well refined in its active time interval. Then, we update v_2's out-neighbors (v_3 and v_4). First we consider v_3. The new time interval for v_3 is $[40 + w(v_2, v_3, 40), 65 + w(v_2, v_3, 65)] = [70, 95]$. Since the previous $C_3(t)$ has no value in [70,95], we use the new one directly. Then, we update v_4 in time interval $[40 + w(v_2, v_4, 40), 65 + w(v_2, v_4, 65)] = [95, 138.75]$. However, since v_4 is a parking vertex, it has to follow the non-increasing property.

In the third iteration, Q becomes $< (\tau_3 = 70, C_3(t)), (\tau_4 = 95, C_4(t)), (\tau_5 = 105, C_5(t)) >$. We pop out top element and visit v_3. The current top is $\tau_4 = 95$ and $w(v_2, v_3, 95) = 30$. So v_3's refined time interval is extended to $[70, 95 + 30] = [70, 125]$, which is larger than v_3's active time interval. So v_3 is also well refined. v_3' out-neighbor v_5's minimum cost function will be computed in time interval $[70 + w(v_3, v_5, 70), 95 + w(v_3, v_5, 95)] = [105, 130]$. τ_5 remains 105. The current Q is $< (\tau_4 = 95, C_4(t)), (\tau_5 = 105, C_5(t)) >$.

In the fourth iteration, we visit v_4 and the top element is $\tau_5 = 105$. $w(v_2, v_4, 105) = 100$ and it extends τ_4 to 205, which exceeds v_4's active time interval, so v_4 is also well refined. We update v_4's out-neighbor v_5 in time interval $[95 + w(v_4, v_5, 95), 125 + w(v_4, v_5, 125)] = [108.75, 130]$. The new $C_5'(t)$ has some lower values compared with the previous one, so we take the lower one as the $C_5(t)$. Finally, the Q has only one element, and we can guarantee that no vertex can update v_5 now. So the minimum on-road travel time from v_1 to v_5 is 100.

4.3.3 Correctness

Before we prove the correctness of *Incremental MORT Algorithm* in Theorem 2, we first prove the minimum cost function is correctly computed. Lemma 3 proves Lines 7–9 is correct, and Lemma 4 proves Lines 10–23.

Lemma 3 *When v_i is popped out and visited, it is guaranteed that $C_i(t)$ will not change in $[\tau_i, \tau_i']$.*

Proof Suppose τ_j is the current top τ in Q. Thus, $\forall \tau_k \in Q, \tau_k \geq \tau_j \Rightarrow C_k(t)$ is well refined before τ_k, which means $\forall v_k \rightarrow v_o, C_o(t)$ has been updated from v_k before τ_k. In other words, no update before time τ_j is possible from now on. The

earliest possible time to update from v_k to v_o is τ_j. Suppose $v_f \rightarrow v_i$, so the earliest possible time to update from v_f to v_i is also τ_j. If we depart from v_f at τ_j, the earliest arrival time at v_i is $\tau_j + w(v_f, v_i, \tau_j)$. Suppose $w(v_f, v_i, \tau_j)$ is the smallest among all in-neighbors of v_i, then the earliest change of $C_i(t)$ will not happen before $\tau_i' = \tau_j + w(v_f, v_i, \tau_j)$. So $C_i(t)$ will not change in $[\tau_i, \tau_i']$. □

Lemma 4 *$C_i(t)$, where $t \in [\tau_i, \tau_i']$, has been updated before it is refined.*

Proof $\tau_i = min\{\tau_j + w(v_f, v_i, \tau_j) | \forall v_i\}$. If v_f is not in Q, then $C_f(t)$ is already refined. So when we finish refining $C_f(t)$, we will update $C_i(t)$ from v_f. If v_f is in Q, then $\tau_f \geq \tau_j \geq \tau_i$. Otherwise, we should have visited v_f earlier than v_i. Thus, v_f's refinement lower bound is no earlier than τ_j, so $C_i(t)$ has been updated from v_f at τ_f, which leads to $\tau_f + w(v_f, v_i, \tau_f) \geq \tau_i'$. Hence, $C_i(t)$ has been updated in subinterval $[\tau_i, \tau_i']$. □

Theorem 2 *Algorithm 3 finds the MORT.*

Proof Lemma 4 guarantees each $C_i(t)$ is fully updated, and Lemma 3 ensures the final $C_i(t)$ is validated incrementally. When v_d's τ_d reaches the latest arrival time t_d, v_d's minimum cost function $C_d(t)$ is fully refined and will not be changed even if the while loop runs on. All the $C_i(t)$ are updated by its in-neighbors, so they are the same as *Basic MORT*'s minimum cost functions. Therefore, the minimum value of $C_d(t)$ is the minimal on-road travel time. □

4.3.4 Complexity analysis

The *ATI* takes $O(|V| \log |V| + |E|)$ time. The initialization phase (Lines 2–5) takes $O(V)$ time. We use *Fibonacci Heap* [48] to implement the priority queue. The size of Q is at most $|V|$, so the extract-min operation on Q takes $O(\log |V|)$ time. Since each vertex v_i's minimum cost function is constructed incrementally, we use L_i to denote the number of its subintervals. Therefore, L_i is actually the number of times v_i would be extracted from Q, which takes $L_i \log |V|$ time. The update and insert on *Fibonacci Heap* take $O(1)$ time, so the maintaining of Q takes $O(\Sigma_{i=0}^{|V|} L_i \log |V|) = O(L(|V| \log |V|))$ time, where L is the average number of subintervals. On the other hand, during the update, we visit all v_i's in-neighbors, which is the same as in-edges E_i^{in}. So if we visit all the in-neighbors of all the vertices, we actually visit every edge. Thus, $\Sigma_{i=0}^{|V|} |E_i^{in}| = |E|$. So the total time complexity is $O(\Sigma_{i=0}^{|V|} L_i (\log |V| + |E_i^{in}|)) = O(L(|V| \log |V| + |E|))$.

Now let's analyze the lower-bound of L_i. Firstly, suppose τ_i^j is the top value in Q and τ_k is the head value, $\tau_i^j \leq \tau_k$. Then $\tau_k + min(w(v_f, v_i, \tau_k)) = \tau_i^{j+1}$, so $\tau_k < \tau_i^{j+1}$. Eventually, we can have a L_i such that $\tau_i^{L_i} \geq v_i.t_d$. Next, we define $\eta_i^0 = v_i.t_s$ and $\eta_i^{j+1} = \eta_i^j + min(w(v_f, v_i, \eta_i^j))$. Eventually, we can get a J_i such that $\eta_i^{J_i} \geq v_i.t_d$. Since for the same j, τ_i^j is always smaller than η_i^j, so we can get $L_i > J_i$. If we use J to denote the average number of J_i, then the lower-bound of L is J. Obviously, $J > T$, so L is also bigger than T.

For the space complexity, the time-dependent parking graph takes $O(|V| + T|E|)$ space. Each minimum cost function $C_i(t)$ takes $O(T)$ space. Q has at most $|V|$ elements, so the size of Q is $O(T|V|)$. Hence, the overall space complexity is $O(T(|V| + |E|))$.

4.4 Application scenarios

In this section, we provide three examples to explain how our algorithm works in different scenarios. It should be noted that the graph structure and time-dependent information are crucial for finding the desired results. Meanwhile, the set of parking vertices and their corresponding minimum waiting time $v.t_{min}$ can also be specified by user depending on different needs.

Firstly, suppose a commuter wants to arrive office faster and depart later. In fact, this is an *ISFP* problem, so we can run our algorithm on a road network that only allows waiting on the departure vertex. Therefore, the departure vertex is the only vertex in the waiting vertex set, and its minimum waiting time is set to 0.

Secondly, suppose a scenario for a truck driver who needs a forced rest every period of time at the service stations along the highway. In this case, the graph is a network of highway, and the parking vertices are some service stations, each has a pre-defined minimum staying time to ensure the rest is sufficient. The traveling time between these stations roughly equals the driver's maximum driving time. Therefore, the force waiting is included in the computation and minimum rest time is guaranteed.

Finally, suppose a traveler is planning a journey from one city to another in several weeks time and wants to visit some of the national parks along his route. In this case, the graph should only contain the national parks as vertices and allows waiting on all of them, which is another extreme case of our model. The graph structure should express the rough traveling order. In this case, it could be organized into a layered graph, and we only visit one of the vertices (national parks) on the same layer. The edges only exist between the vertices in neighboring layers. In an extreme case when the traveler wants to visit every park, the graph should be organized as a linear line. It should be noted the graph structure can also reflect the distribution of waiting schedule. We can set the distribution of parking vertex manually to meet users' waiting requirement (e.g., set the corresponding minimum waiting time to ensure a forced rest every period of time or a minimum visiting time). Next, we should not use the traffic condition as the only parameter to determine the time-dependent weight

functions. In fact, the functions should take both travel cost and drivers' willingness into account. For instance, it is a journey rather than a hurrying on the way, so we should avoid the unsafe night driving. Thus, the weights during night time should be set much higher even though the traffic condition is good. In fact, all the weights for the time that are not suitable for driving, either due to bad traffic condition or due to travelers' preference, should be set higher. After that, our algorithm could find a *MORT* traveling schedule on this time-dependent graph.

5 α-MORT approximation

In this section, we present several approximation methods to solve the *MORT* problem faster with a guaranteed lower bound α. As analyzed in Sect. 4.2.4, the time complexity is significantly affected by the number of turning points in $C_i(t)$. What is worse, it grows larger as the expansion grows, which makes the computation slower and slower. So the key to speed up is decreasing the number of turning points, especially the useless ones. However, we cannot determine if one turning point will end up with the optimal result until the final $C_d(t)$ is constructed. Therefore, we design an approximation approach that can guarantee the final result is no less than $\alpha C_d^*(t)$, $\alpha \in (0, 1]$. Section 5.1 introduces the approximation error α and how the error grows as the route grows. Sections 5.2–5.4 describe three approximate methods in detail.

5.1 Error bound α and turning point pruning

Given an input approximation ratio α, we aim to compute a route whose *MORT* time $A_d^*(t) \geq \alpha C_d^*(t)$. However, the approximation cannot be applied on each edge along the route directly.

Suppose a route is made up of a series of consecutive edges $\hat{E} =< e_1, e_2, \ldots, e_n >$ and $||\hat{E}||$ is the length of $\hat{E}$. If we apply an approximation factor α_1 on e_1, α_2 on e_2 and so on, the error of the final result does not grow linearly, as shown below:

$$\begin{aligned}
||\hat{E}||' &= ((((e_1\alpha_1 + e_2)\alpha_2) + e_3)\alpha_3 + \cdots e_n)\alpha_n \\
&= \alpha_1\alpha_2\alpha_3\cdots\alpha_n\ e_1 + \alpha_2\alpha_3\cdots\alpha_n e_2 + \cdots + \alpha_n e_n \\
&= \prod_{j=1}^{n}\alpha_j e_1 + \prod_{j=2}^{n}\alpha_j e_2 + \cdots + \prod_{j=n}^{n}\alpha_j e_n \\
&= \sum_{i=1}^{n}\prod_{j=i}^{n}\alpha_j e_i
\end{aligned}$$

To achieve $||\hat{E}||' \geq \alpha||\hat{E}||$, we have to guarantee $\prod_{j=1}^{n}\alpha_j \geq \alpha$. In another word, we can view α as a total budget of pruning power along the route, the larger the budget assigned to a vertex, the stronger pruning power it has to reduce the turning points. Because the turning point numbers of the earlier visited vertices are much smaller than those of the latter visited ones, we concentrate the pruning power to the latter vertices by setting a global turning point number threshold ρ: Only those vertices whose turning point numbers are larger than ρ will be pruned. Because in the *ATI* computation we already have two functions of *earliest arrival path* and *latest departure path*, we use $\frac{min(|EA(v_d)|,|LD(v_s)|)}{3}$ as a heuristic threshold value, where $|EA(v_d)|$ and $|LD(v_s)|$ are the turning point numbers of those two paths. The details of how to assign pruning power α_j are discussed from Sects. 5.2 to 5.4.

Algorithm 4: $C_i(t)$ Pruning Algorithm

```
   Input: C_i(t) = (p_1, p_2, ..., p_n), pruning power α_i
   Output: α_i-approximate A_i(t)
1  begin
2      i ← 2
3      while i ≤ |C_i(t)| − 1 do
4          PointList.insert(p_i)
5          for p_j ∈ PointList do
6              //compute p_{j,i+1} on (p_{i−1}, p_{i+1})
7              p_{j,i+1} ← Compute(p_{i−1}, p_{i+1}, p_j)
8              if p_{j,i+1} ≥ α_i p_j and p_j ≥ p_{j,i+1} then
9                  i ← i + 1
10                 PointList.clear()
11                 break
12         C_i(t).prune(p_i)
13     return A_i(t) ← C_i(t)
```

At this stage, we assume $|C_i(t)| > \rho$ and it has a pruning power α_i. The pruning process visits the turning points one by one in a sliding window way, as shown in Algorithm 4. Each time we visit a turning point p_i, we put it into a *PointList* (Line 4). It can be pruned only if all the points p_j in *PointList* can be safely represented by point $p_{j,i+1}$ on line (p_{i-1}, p_{i+1}) (Line 7). The safe representation has two conditions (Line 8). Firstly, $p_{j,i+1}$ has to be no smaller than $\alpha_i p_j$, as required by approximation bound. Secondly, p_j is no smaller than $p_{j,i+1}$, because the smaller value has a higher possibility to result in the final optimal result. If any p_j does not satisfy these two conditions, p_i cannot be pruned and we empty the *PointList*. When all the points are visited, we return the remaining points as the approximate function $A_i(t)$. Since $p_{j,i+1} \geq \alpha_i p_j$ is strictly required, $A_i(t) \geq \alpha_i C_i(t)$. In the worst case when all the points within $C_i(t)$ is pruned, the testing in line 8 has to run $O(|C_i(t)|^2)$ times. However, it has a near linear running time in practice.

Figure 7 shows a pruning example. It should be noted that although the pruning procedure looks similar to the trajectory compression/segmentation, it does not run on the final $C_i(t)$ for each vertex because we cannot get them until the expan-

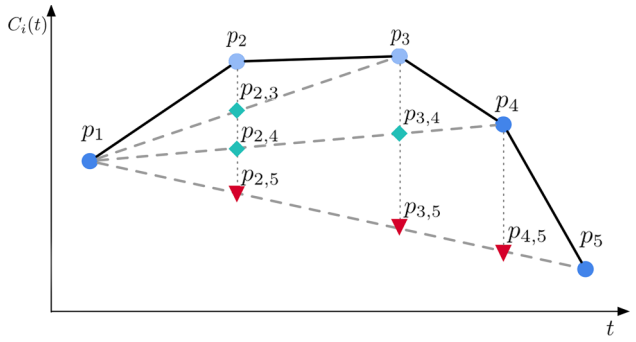

Fig. 7 $C_i(t)$ Turning points pruning example. p_2 can be pruned because its new point $p_{2,3}$ represented by line (p_1, p_3) is larger than $\alpha_i p_2$. p_3 can be pruned because $p_{2,4} > \alpha_i p_2$ and $p_{3,4} > \alpha_i p_3$. But p_4 cannot be pruned because their new values on the new approximate line (p_1, p_5) are smaller than $\alpha_i p_2$, $\alpha_i p_3$ and $\alpha_i p_4$

sion finished. The true contribution of the approximation lies in the pruning power distribution approaches, as described in the following sections.

5.2 Even distribution

The first way to assign pruning power is distributing them evenly. Suppose $|\hat{E}|$ is number of edges in route $\hat{E}$. The most straightforward way is to assign $\alpha_i = \alpha^{\frac{1}{|\hat{E}|}}$, as shown below:

$$\sum_{i=1}^{n}\prod_{j=i}^{n}\alpha_j e_i = \sum_{i=1}^{n}\prod_{j=i}^{n}\alpha^{\frac{1}{|\hat{E}|}} e_i = \sum_{i=1}^{n}\alpha^{\sum_{j=i}^{n}\frac{1}{|\hat{E}|}} > \alpha\sum_{i=1}^{n} e_i$$

However, pruning power $\alpha^{\frac{1}{|\hat{E}|}}$ becomes weaker when $|\hat{E}|$ is larger, which makes the pruning insufficient. Therefore, we restrict the pruning power α shared by only $|\check{E}|$ vertices along the route, where $|\check{E}| \ll |\hat{E}|$. Thus, the vertices has larger pruning power when their turning point numbers surpass the threshold ρ.

5.3 Exponential distribution

The second way to distribute pruning power is decreasing the power exponentially. In this way, the first pruning vertex can have a much larger power than those in then even distribution. We assign $\alpha_1 = \alpha^{\frac{1}{2}}$, $\alpha_2 = \alpha^{\frac{1}{2^2}}$ and so on. In this way, the approximate bound is guaranteed, as proved below:

$$\begin{aligned}||\hat{E}||' &= \alpha_1\alpha_2\alpha_3\cdots\alpha_n\, e_1 + \alpha_2\alpha_3\cdots\alpha_n e_2 + \cdots + \alpha_n e_n \\ &= \alpha^{\frac{1}{2}}\alpha^{\frac{1}{2^2}}\cdots\alpha^{\frac{1}{2^n}} e_1 + \alpha^{\frac{1}{2^2}}\cdots\alpha^{\frac{1}{2^n}} e_2 + \cdots + \alpha^{\frac{1}{2^n}} e_n \\ &= \alpha^{\sum_{i=1}^{n}\frac{1}{2^i}} e_1 + \alpha^{\sum_{i=2}^{n}\frac{1}{2^i}} e_2 + \cdots + \alpha^{\sum_{i=n}^{n}\frac{1}{2^i}} e_n \\ &> \alpha e_1 + \alpha^{\frac{1}{2}} e_2 + \cdots + \alpha^{\frac{1}{2^n}} e_n > \alpha\sum_{i=1}^{n} e_i = \alpha||\hat{E}||\end{aligned}$$

Although the pruning is large in the beginning, it decreases fast as the pruned vertices grows. So similar to the *Even Distribution*, we also set a small upper bound of n to avoid useless pruning.

5.4 Dynamic exponential distribution

The previous two methods assign fixed pruning power to each vertex and do not care whether the power is fully utilized or not. In fact, most of the vertices only use part of their power, which is a waste of the precious budget. In order to take the most advantages of the precious pruning budget, we propose the *Dynamic Exponential Distribution* method.

Like the *Exponential Distribution*, the pruning power also decreases exponentially. However, instead of dividing the previous pruning power's logarithm by 2, we divide the remaining pruning power's logarithm by 2. Initially, the algorithm keeps pruning power's logarithm Δ_i for each vertex and set them to 1. The first pruning vertex v_k has the pruning budget $\alpha^{\frac{\Delta_k}{2}} = \alpha^{\frac{1}{2}}$. During the pruning, we can get its actual pruning usage by $\beta = max(p_{i,j}/p_i)$, where p_i is the pruned point. Then, the remaining pruning power logarithm for $v_k's$ out-neighbor v_j is $\delta_k - log_\alpha\beta$. If v_j is to be pruned, its pruning budget is $\alpha^{\frac{\delta_k - log_\alpha\beta}{2}}$. We also set a lower bound for α_i to avoid the useless pruning.

The proof of bound guarantee is similar to *Exponential Distribution*.

6 Speed profile generation

In this section, we explain how we derive the speed profile from the trajectory data. We first talk about how to obtain the road speed from the trajectory. Then, we present our observations on the effects of different granularities of the speed collections. After selecting an appropriate time slot size, we use several approaches to estimate missing values. Finally, we test three compression algorithms on our speed profiles in order to reduce the storage space.

6.1 From trajectory to road speed

First of all, we match the trajectory $tr_i =< (x_1^i, y_1^i, t_1^i), \ldots, (x_m^i, y_m^i, t_m^i) >$ to the graph G. There are several methods [49–51] in this field. After that, we obtain a sequence of consecutive edges $E_i =< e_1, \ldots, e_m >$, with $\forall 1 \le j \le m$, (x_j^i, y_j^i, t_j^i) is on some edge $e_k \in E_i$. It should be noted that an edge could have several points attached to it, while some edges might have no attaching points. For any consecutive pair of points $p_j^i = (x_j^i, y_j^i, t_j^i)$ and $p_{j+1}^i = (x_{j+1}^i, y_{j+1}^i, t_{j+1}^i)$, we can retrieve a set of edges $E_j^i =<$

$e_k, e_{k+1}, \ldots, e_n >$ between them. We assume the travel between p^i_j and p^i_{j+1} keeps an even speed. The distance $d^i_{j,j+1}$ between them is the sum of corresponding traveled edge length. Thus, the speed is $v^i_j = d^i_{j,j+1}/(t^i_{j+1} - t^i_j)$. Then, we attach this speed to the corresponding edges in E^i_j, with time proportional to the distance to p^j_j. By repeating this procedure from the first GPS point to the last, we can get all the roads' speeds along this trajectory, together with their corresponding time.

6.2 Speed data collection

Before collecting the data into time slots, we first categorized them into weekday and weekend, or by date. Then, for each edge e_i in one specific category, it has a set of speed data $< (v^i_1, t^i_1), \ldots, (v^i_n, t^i_n) >$. The next step is converting it into a usable speed profile.

The most straightforward method is to use these speed data directly, which would result in a set of linear piecewise speed functions. However, it is not practical for the following reasons. Firstly, some speeds are either much smaller than the others because the driver may wait for the traffic light or even stop to wait for a passenger, or bigger than the average due to some emergency cases. If we line up these speed points directly, we will get a zigzag speed profile that apparently cannot describe the road network's actual traffic condition. In fact, it falls into the terrible situation of overfitting. Secondly, a speed profile with a random bunch of functions is both hard to use and compress. Another approach is approximating the speed data using some regression methods [52,53]. Although it can represent the speed profile as functions, it is unable to deal with missing value since it estimates the missing speed only by the values on each edge itself, which is highly inaccurate.

To address the problems mentioned above, we use a histogram-based approach to collect the speed data. Specifically, we divide one day's time into T slots with the same length. Then, the speed data that fall into the same slot will be added up together to get an average speed. Thus, the influence of the outliers is reduced dramatically. However, the granularity of the histogram is another important issue to consider. If T is small, it cannot reflect the difference of traffic conditions during different time of a day. While if T is big, there will be not enough speed data within each time slot and the size of the speed profile will soar up at the same time. We test the granularity of *1-day, 1-h, 30, 15 and 5-min* in Sect. 7.4.1. Based on the experiment results, we choose the 5-min time slot, whose number is 288 for each edge in a day, to collect the speed.

6.3 Missing value estimation

Even though the GPS-based trajectory data has a higher coverage of the road network than other approaches, it is still hardly possible to cover every edge. So, it also faces the sparsity problem. To make the matter worse, the data become even sparer as the number of time slots grows. In our test, although the 5-min granularity is not too small to produce too many void time slots, there are still 85% of them have no value. In this section, we propose two approaches to estimate the missing values in the histogram data: *Cosine Similarity* and *Spatial-Temporal Neighboring Average*.

6.3.1 Cosine similarity

This approach compares the similarity between an edge and its neighbors and uses the similar ones' data to fill its missing values. Each road e_i's speed profile can be viewed as a speed vector SP_i with T values: $SP_i =< SP_{i,0}, SP_{i,1}, \ldots, SP_{i,T-1} >$, where $SP_{i,j}$ is the speed of edge e_i at time slot j. If there is a missing value, we just use 0 to denote it. $|SP_i|$ denotes the number of time slots without missing values. Thus, the similarity between the speed profiles of two edges e_i, e_j can be evaluated by the cosine similarity:

$$\begin{aligned}\text{Coorelation}\,(SP_i, SP_j) &= \frac{SP_i \cdot SP_j}{\| SP_i \| \| SP_j \|} \\ &= \frac{\sum_{k=0}^{T-1} SP_{i,k} \times SP_{j,k}}{\sqrt{\sum_{k=0}^{T-1} (SP_{i,k})^2} \times \sqrt{\sum_{k=0}^{T-1} (SP_{j,k})^2}}\end{aligned}$$

We use $SP_i \cap SP_j = \{k | SP_{i,k} \neq 0 \wedge SP_{j,k} \neq 0\}$ to denote the time slots that are not empty on both edges. Furthermore, in order to eliminate the bias from the edges with sparse speed profile, we calculate the similarity only when $|SP_i \cap SP_j| > 25\% \times T$. For each edge, we compute its similarities between its 3-hop neighboring edges and find the top-3 similar ones. Then, it uses the speed in these three profiles to fill its missing speeds. For a specific time slot, if the most similar one is also empty, then we check the second most similar one. If still empty, then check the third.

The computation works iteratively from the edges with higher $|SP_i|$ to lower ones. As the process proceeds, the $|SP_i|$ changes at the same time. Eventually, the edges with $|SP_i|$ larger than $25\% \times T$ would get fully filled. For those sparse ones, we apply the *Spatial-Temporal Neighboring Average* approach described in Sect.6.3.2.

6.3.2 Spatial-Temporal Neighboring Average

This is the simple approach that averages the speed of a road's neighbor and its neighboring time slots.

$$SP_{i,j} = \text{Avg}(SP_{k,j}, SP_{i,j-1}, SP_{i,j+1}), \quad \forall e_k \cap e_i \neq \phi$$

where $SP_{i,j}$ is edge e_i's speed at its jth time slot. If its neighbors are also empty at certain time slots, we extend the search to the 3-hop neighbors and 3-hop time slots. The computation also computes iteratively starting from the roads that have fewer missing values. This is because these roads always link to roads that have a relatively complete speed profile. Then, it propagates all the roads in the road network eventually.

6.4 Speed profile compression

As mentioned previously, the smaller the time slot size, the less space-efficient the speed profile is, especially when the neighboring slots have the same or similar speeds. To save the space for storing the speed profiles on disk and in memory, we propose an *adaptive speed profile*. The term *adaptive* means this speed profile is derived from the histogram-based profile and adapts the occasions where the nearby time slots have similar speed values. In this subsection, we aim to reduce the speed profile size from the perspective of each road. We test three different kinds of *Piecewise Linear Approximation* [54,55] algorithms to convert the 5-min-histogram-based speed profile to a set of piecewise linear functions. The actual speed of each road at different time can be computed by the corresponding function.

The histogram-based speed profile can be viewed as a type of *Time Series Data* [56], and building the adaptive speed profile from the histogram-based speed profile falls into the category of *Time Series Segmentation* and is defined as below:

Definition 2 (*Speed Profile Segmentation*) Given a speed time series $SP_i =< SP_{i,0}, SP_{i,1}, \ldots, SP_{i,T-1} >$, construct a model $\hat{SP}_i =< SP_{i,0}, \ldots, SP_{i,\hat{d}} >$ of reduced dimensionality $\hat{d}$, ($\hat{d} \ll T-1$) such that $R(\hat{SP}_i, SP_i) < \varepsilon$, where R is a reconstruction function and ε is a given error threshold.

The reconstruction function R calculates the difference of speed value between adaptive speed profile and the original one. It serves as the evaluation method of compression quality. We choose the *residual error* as the reconstruction function, which adds up the square of the differences. *PLA (Piecewise Linear Approximation)* [54] is the compression approach which aims at transferring the original SP_i into a set of approximate lines while retaining the essential features. There are two ways of approximation:

- *Linear Interpolation* Use a line connecting the two ending points to approximate.
- *Linear Regression* Use the linear regression algorithms to find the best fitting line.

Apparently, linear interpolation has a smooth look, while linear regression produces a set of dis-joint segments. We choose linear interpolation approach because of the following reasons. Firstly, it is obviously faster to implement and compute. Secondly, it is more space saving than linear regression. Linear interpolation only needs to store the turning points, while linear regression has to store all the end points of the segments, which is twice larger. Moreover, since the speeds in profile are all approximate, the more accurate algorithm in this step cannot promise a better approximation.

There are three basic categories of *PLA*: *sliding window* [54], *top-down* [57] and *bottom-up* [58]. They are described in the following sections:

6.4.1 Sliding window algorithm

The sliding window algorithm is a fast online algorithm whose time complexity is $O(n)$, where n is the speed profile length of an edge. It keeps expanding the approximate line from the left starting point to the right until the *error* surpasses a user-specified threshold ε. Then, it uses the end point of the last generated segment as the next starting point and repeats until all the points are visited. Since each point in the speed profile is visited only once, it has a linear complexity. The detail is shown in Algorithm 5. $SP_{i,j,k}$ denotes the speed profile segment of edge e_i from its jth speed point to kth speed point.

Algorithm 5: Sliding Window Algorithm

Input: The speed profile of an edge $SP_i =< SP_{i,0}, ..., SP_{i,T-1} >$, error threshold ε
Output: The adaptive speed profile of a road $\hat{SP}_i^{sw} =< SP_{i,0}, ..., SP_{i,\hat{d}} >$

```
1  begin
2      SP^sw_i.insert(SP_{i,0})
3      for j from 0 to T−1 do
4          for k from j + 1 to T−1 do
5              if R(SP_{i,j,k} SP^sw_{i,j,k}) > ε then
6                  SP^sw_i.insert(SP_{i,k})
7                  j = k − 1
8                  break
9      if SP_{i,T−1} not in SP^sw_i then
10         SP^sw_i.insert(SP_{i,T−1})
11     return SP^sw_r
```

6.4.2 Top-down algorithm

The top-down algorithm finds the best speed point that splits the original speed profile into to segments each time (i.e., where the two resulting segments have the smallest combined error). If the any of the two resulting segment's error is larger than threshold ε, the algorithm repeats recursively to find the best splitting speed points in it. The algorithm terminates when all the speed profile segments' errors are smaller than ε.

It breaks the search space into two pieces each time and calls for itself recursively at most twice. At the same time, the R function calculates the difference between the result speed profile segment and the original one, which takes $O(n)$ times. So the overall time complexity of the top-down algorithm is $O(n \log n)$. As the threshold ε grows, less recursion is needed, and the overall running time decreases.

Algorithm 6: Top-down algorithm

Input: The speed profile of an edge $SP_i =< SP_{i,0}, ..., SP_{i,T-1} >$, error threshold ε
Output: The adaptive speed profile of a road $\hat{SP_i^{td}} =< SP_{i,0}, ..., SP_{i,\hat{d}} >$

```
1  begin
2      Function TDFindBreakPoint(int low, int high)
3          best_so_far = inf
4          for j from low + 1 to high − 1 do
5              BestTmp = R(SP_{i,low,j}, ŜP^{td}_{i,low,j}) +
               R(SP_{i,j,high}, ŜP^{td}_{i,j,high})
6              if BestTmp < best_so_far then
7                  best_so_far = BestTmp
8                  k = j
9          ŜP_i^{td}.insert(SP_{i,k})
10
11         if R(SP_{i,low,k}, ŜP^{td}_{i,low,k}) > ε then
12             TDFindBreakPoint(low, k)
13         if R(SP_{i,k,high}, ŜP^{td}_{i,k,high}) > ε then
14             TDFindBreakPoint(k, high)
```

6.4.3 Bottom-up algorithm

The bottom-up algorithm is reverse to the top-down algorithm. In the initial step, it connects the points in the original speed profile, so errors are all 0. After that, it merges consecutive lines, by erasing the intermediate point, with the smallest error iteratively until the smallest error exceeds the threshold ε. In the worst case, we have to erase all the intermediate points, which runs $\frac{n(n-1)}{2}$ times. Therefore, the time complexity of bottom-up algorithm is $O(n^2)$. The detail is shown in Algorithm 7.

Algorithm 7: Bottom-Up Algorithm

Input: The speed profile of an edge $SP_i =< SP_{i,0}, ..., SP_{i,T-1} >$, error threshold ε
Output: The adaptive speed profile of a road $\hat{SP_i^{bu}} =< SP_{i,0}, ..., SP_{i,\hat{d}} >$

```
1  begin
2      ŜP_i^{bu} =< SP_{i,0}, ..., SP_{i,T−1} >
3      do
4          minError = inf
5          for j from 1 to T − 1 do
6              errorTmp = R(SP_{i,j−1,j+1}, SP^{bu}_{i,j−1,j+1})
7              if errorTmp < minError then
8                  minError = errorTmp
9                  bp = SP_{i,j}
10         if minError < ε then
11             ŜP_i^{bu}.erase(bp)
12     while minError ⩽ ε
```

7 Experiments

In this section, we first describe the experiment setup in terms of datasets, online query setup and speed profile evaluation metrics. After that, we present the result of a comprehensive performance study to demonstrate the effectiveness and efficiency of our *MORT* algorithms. Finally, we show the experiment results of the offline speed profile generation.

7.1 Experiment setup

7.1.1 Datasets

We first describe the map datasets we use for the whole system. Then, we present the trajectory data we use for the speed profile generation.

We get two maps of Beijing and Shanghai from *Navinfo*.[1] The Beijing map consists of 302,364 intersections and 387,588 roads, which covers a 184 km×185 km spatial range and has a total length of 51,666 km of roads. The road network of Shanghai has 243,842 intersections and 310,058 roads. It covers a 120 km×143 km spatial range. The total length of road segments is 42,930 km. As for the parking vertices, we attach them on maps randomly to test its influence on the algorithms.

We obtain our trajectory data from *DiDi*.[2] It has the trajectory data of five consecutive days from 2015.4.1 to 2015.4.5, collected from taxis in Beijing and Shanghai, respectively. The total data set has 2,171,882 trajectories and 74,948,829 GPS points in Beijing, and 1,402,047 trajectories and 40,203,623 GPS points in Shanghai. The average sampling rate is 5 s. 89% of the roads in Beijing and 75% of the road in Shanghai are covered by the trajectory. The details

[1] http://www.navinfo.com/.

[2] http://www.xiaojukeji.com/news/newslisten.

Table 2 Trajectory data sets

City	Num	4.1	4.2	4.3	4.4	4.5
Beijing	Traj	532,868	143,998	541,650	310,976	642,390
	GPS	17,698,668	5,164,315	17,069,156	11,402,483	23,614,206
Shanghai	Traj	389,733	103,411	378,968	180,670	349,265
	GPS	10,747,519	2,949,734	10,039,956	5,519,847	10,946,567

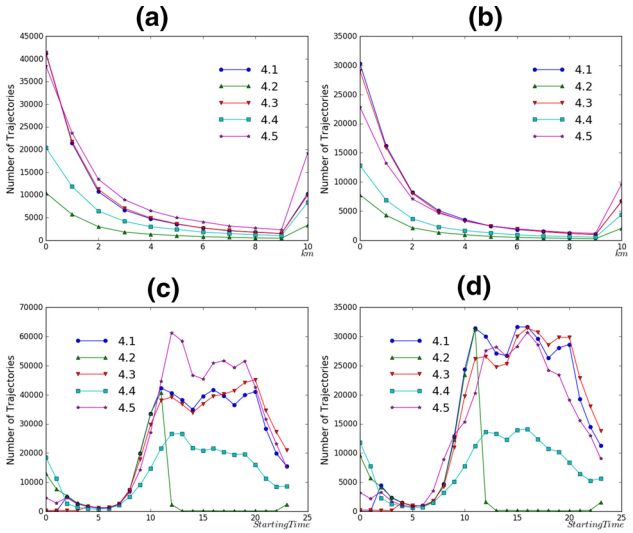

Fig. 8 Trajectory starting time and length distribution

of each day's basic information are shown in Table 2. The trajectory's length distribution of each city on each day is present in Fig. 8a, b. It shows that the number of trajectory decreases as the length grows, so most of our trajectories are not too long. As for the last value point that soars up, that is because it is the accumulation of the all the trajectories that have length no shorter than 10 km. The starting time distribution along 24 h is shown in Fig. 8c, d. Except for 2015.4.2, which lacks some data, most trajectories are collected during daytime and few trajectory appears after midnight. This distribution corresponds to the people daily behavior, and we build our test speed profile on daily basis.

7.1.2 MORT experiment setup

We test the algorithms under four variations. The first one is the distance of two vertices on road network. The second one varies the starting time interval size from 1, 2, 3 to 4 h. The next one tests the performance under different speed profiles (50, 100, 200, 400 turning points), and the last one varies the percentage of parking vertices (5, 10, 50, 100%). Except for the third test, which uses synthetic speed profile, all the experiments use the speed profiles generated from the trajectory data.

7.1.3 Speed profile evaluation metrics

We use 80% of the trajectories that are selected randomly to build the speed profiles and test them on the remaining 20% trajectories. In the evaluation, we re-travel the testing trajectories using the speed from the generated speed profile since these trajectories are the only ground truth we have. For any trajectory Tr_i, we first match it on map and convert it into a sequence of consecutive edges like $< Tr_i.e_0, Tr_i.e_1, \ldots, Tr_i.e_k >$, which starts on time t_0 and stops on t_{k+1}, its average speed along this trajectory is

$$Tr_i.\text{speed} = \frac{\Sigma_{j=0}^{k} Len(Tr_i.e_j)}{t_{k+1} - t_0} \tag{1}$$

Then, we re-travel this trajectory by following exactly the same roads in the same order from t_0 using the testing speed profile, and it will finish traveling $Tr_j.e_k$ on t'_{k+1}. The new average speed is

$$Tr_i.\text{speed}' = \frac{\Sigma_{j=0}^{k} Len(Tr_i.e_i)}{t'_{k+1} - t_0} \tag{2}$$

Then we can calculate the *mean absolute error (MAE)* of each speed profile as

$$MAE = \frac{\sum_{i=0}^{N} |Tr_i.\text{speed} - Tr_i.\text{speed}'|}{|Tr|} \tag{3}$$

where $|Tr|$ is the number of testing trajectories. The smaller the *MAE*, the better the speed profile. The unit of *MAE* is m/s. During the test, we omit those trajectories that are short since they are more easily affected by the abnormal driving behavior, while the longer ones suffer less from it.

7.1.4 Experiment environment

We ran all the experiments on a Dell R720 PowerEdge Rack Mount Server which has two Xeon E5-2690 2.90 GHz CPUs, 192 GB memory, 1 TB hard disk, and runs Ubuntu Server 14.04 LTS operating system.

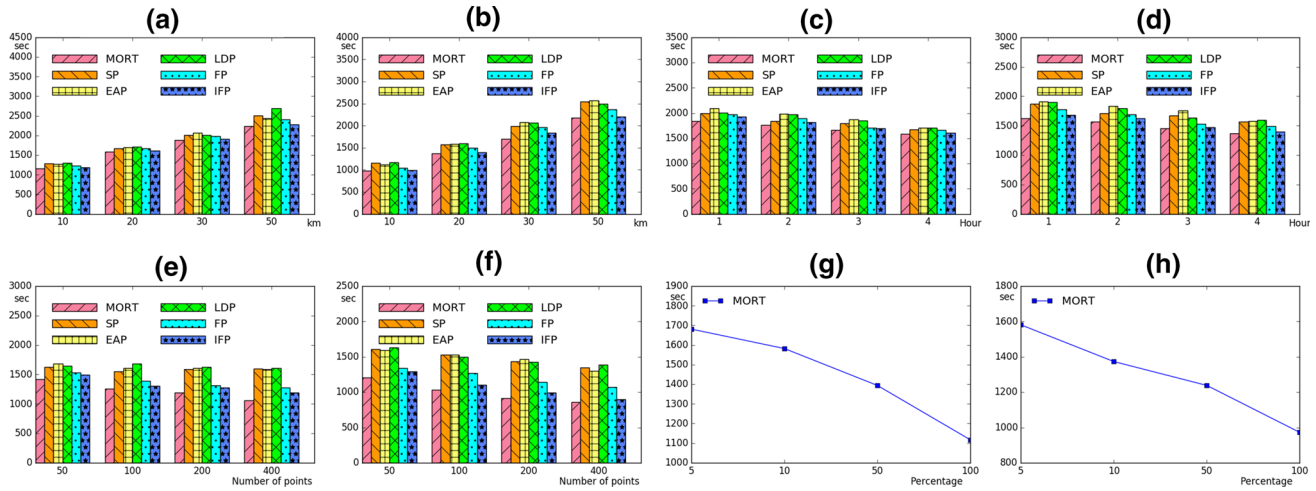

Fig. 9 Results of minimal on-road time

7.2 MORT algorithms evaluation

7.2.1 Comparison with existing algorithms

In this section, we compare the minimal on-road time routes computed by our algorithm with paths generated by the other path planning algorithms under different configurations. We compared our methods with the following algorithms: (1) *SP (Shortest Path)* which computes the shortest path between two vertices. We set the departure time randomly within the time interval. (2) *EAP (Earliest Arrival Path)* and *LDP (Latest Departure Path)*, which are two bypass results when computing the minimal on-road time. (3) *FP (Fastest Path)* [5]. (4) *IFP (Iterative Fastest Path)* which uses the *FP (Fastest Path)* algorithm iteratively to get the approximate minimal on-road time route, as described in Sect. 1. The results achieved by our algorithms are labeled with *MORT*. We do not distinguish the two versions of our algorithms in this experiment since they produce the same on-road travel time.

In the first test, we change the distance between v_s and v_d. We randomly select four sets of vertex pairs with the approximate distance of around 10, 20, 30, and 50 km in the two maps. The starting time interval is set to be 4 h. 10% of the vertices are selected as parking vertices. The results are shown in Fig. 9a, b. It is obvious that our algorithms always produce the shortest on-road travel time, followed by *IFP* and *FP*. As for the other three algorithms, they do not have a chance to achieve a shorter on-road time by changing the departure or waiting time, so their performance is unstable and worse than the previous algorithms in average.

The second test varies the length of starting time interval from 1 to 4 h. The distance is set to be 20, speed profile is 100, and parking vertex is 10%. Figure 9c, d shows the results. As the length of the time interval grows, more possible starting time emerge, so the on-road time of *FP* and *IFP* decreases. As for *MORT*, it also decreases because it has a longer time to wait for a faster route on the parking vertices. And it decreases faster than *FP* because it can get more benefits. As for the other algorithms, they do not change much correspondingly due to the same reason as the previous test.

The third test evaluates the influence of the speed profile granularity, whose turning point numbers are 50, 100, 200 and 400. The distance is also 20, parking vertex is 10%, and the starting time interval is 4 h. We can see from Fig. 9e, f that as the total number of turning points grows, the number of the turning points that have smaller traveling cost also increases. So, there is a higher chance for *FP* and *IFP* to find routes with smaller on-road time. And *MORT* also decreases more distinctly for the same reason.

The last test studies the influence of the park vertex percentage, which varies from 5, 10, 50 to 100%. The distance is 20, the speed profile has 100 turning points, and time interval is 4 h. Figure 9g, h only shows the on-road time of *MORT* because the results of all the other methods do not change along with the percentage of parking vertices. It is easy to draw the conclusion that as the percentage rises, the on-road time drops accordingly since it has more vertices able to wait for a shorter on-road time.

7.2.2 Algorithm running time

In this section, we compare the running time of our algorithms on the three graphs under the same setting of the previous experiments. Apart from the running time of our *Basic* and *Incremental* algorithms, we also show the performance of *IFP* in the first test, and *Fastest Path* in the second and third tests.

Firstly, Fig. 10a, b shows the results under different distances. As the distance grows, the numbers of the visited

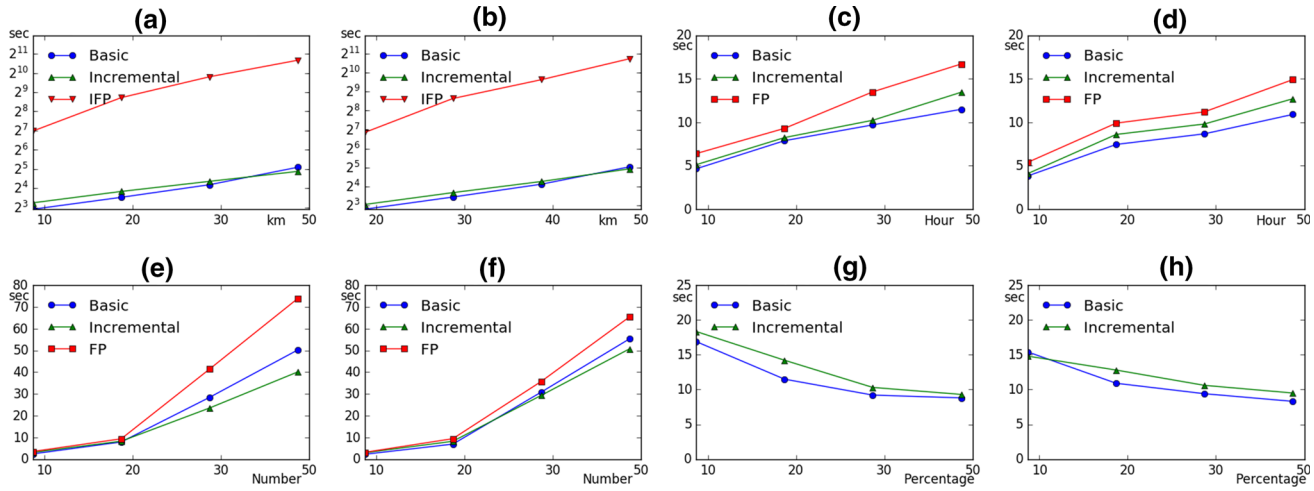

Fig. 10 Algorithm running time

vertices and edges also grow, so the running time increases. Not surprisingly, the running time of *IFP* soars up, so we demonstrate it in exponential step. Secondly, the impact of time interval is illustrated in Fig. 10c, d. As the interval grows longer, the active time interval also grows, which makes the minimum cost function longer. Both algorithms run slower because more turning points appear in the minimum cost functions.

Furthermore, we demonstrate the running time on different speed profiles in Fig. 10e, f. If the density of the speed profile rises, the number of the turning points in the minimum cost function also increases. However, different from the growth of the time interval, which increases the turning points linearly, the growth of time points in speed profile raises the point number in minimum cost functions more dramatically. And the *Basic* algorithm has higher cost on maintaining larger cost function, so it becomes slower than the *Incremental* algorithm. In addition, as shown in Fig. 10c, f, *FP* is always slower than *MORT*. The reason is that *FP* cannot apply *non-increasing*, so it always has more turning points in the minimum cost functions.

Finally, we present the influence of the percentage of parking vertices in Fig. 10g, h. Since the minimum cost function of a parking vertex is non-increasing, the number of its turning point is smaller than the ordinary vertices. Therefore, as the percentage of the parking vertices increases, the total number of the turning points decreases. So the running time drops correspondingly. We do no present the running time of *FP* because its running time is not affected by the parking vertices.

Even if our algorithms are faster than the state-of-art fastest path algorithm, it is still slow for the long distance query. So we will present an index to answer the time-dependent path queries under a second in the future work. But algorithms in this paper are the basis for the index.

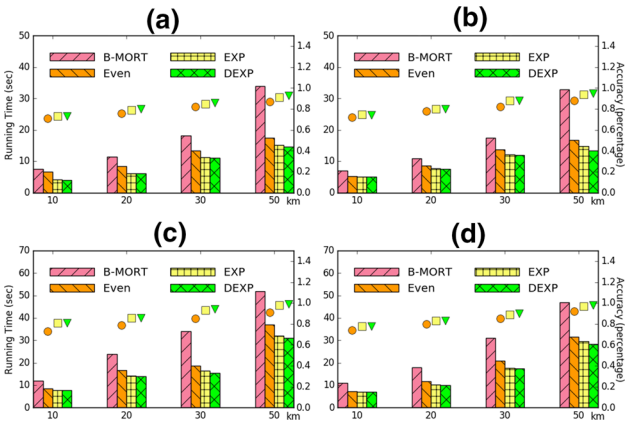

Fig. 11 Running time and accuracy of α-MORT

7.3 Approximation algorithm

In this section, we test the running time and accuracy of our approximation algorithms on two road networks. The results are shown in Fig. 11. As analyzed in Sect. 5.1, the error decreases as the route grows longer. In fact, we can still get a good approximation result even if the initial error bound is small. We show the results of $\alpha = 0.2$ in this test. First of all, as the distance grows, the approximation performs better. For example, the running time is nearly half of the original algorithm in the 50 km test, while the accuracy is around 97%. This is because the longer routes have much more turning points than the short ones, and pruning those points could lead to more benefit. And the pruning power are the same for all the routes regardless of their lengths, so the longer routes have higher accuracy. Secondly, even though the *Even Distribution* can reduce the running time dramatically, the *Exponential Distribution* has an even better performance, while the *Dynamic Exponential Distribution* is

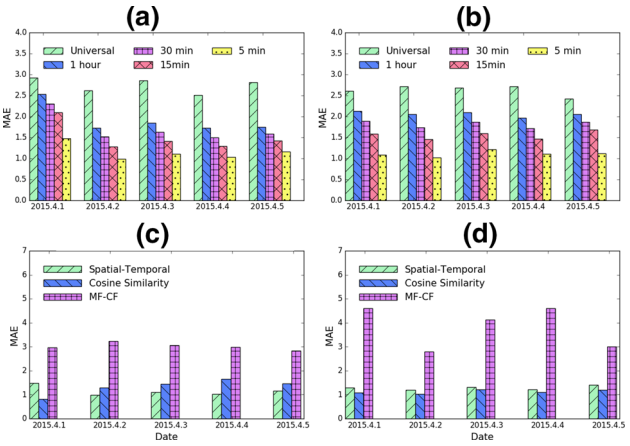

Fig. 12 *MAE* of speed profiles under different granularity and using different missing value estimation

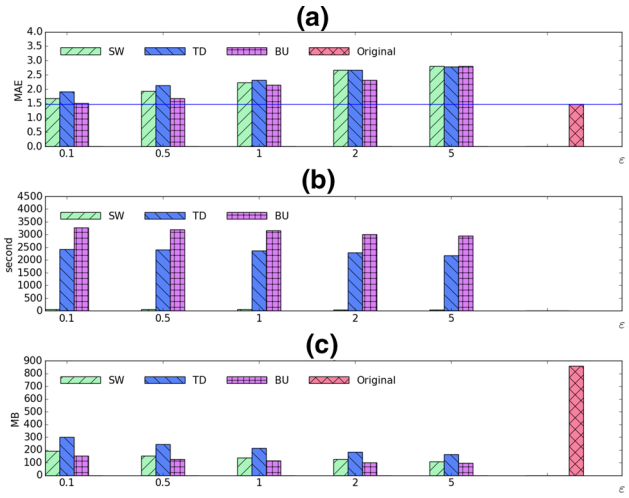

Fig. 13 Compression performance on Beijing map 2015.4.1

only slightly better the *Exponential Distribution*. The reason of it is that although *Dynamic Exponential* makes better use of the pruning budget, its dynamic mechanism takes extra costs. Finally, the percentage of parking vertices also affects the approximation performance. The no parking tests have higher accuracy and speedup. The reason is the same as the distance: less parking vertices along the route results in more turning points.

7.4 Speed profile generation evaluation

7.4.1 Granularity

We compare the *MAE* of speed profiles under granularities of 1-day (Universal), 1-h, 30, 15 and 5-min on 5 days in Beijing and Shanghai, respectively. The results are shown in Fig. 12a, b. We can observe clearly that the 5-min speed profile outperforms the others. The *MAE* increases as the time slot size grows. The universal granularity, which is actually a static graph, has the largest *MAE* obviously. So for the rest of the tests, we only present the results of the 5-min speed profile.

7.4.2 Missing value estimation

We compare three missing value estimation approaches in this test: *Cosine Similarity*, *Matrix-Factorization-based Collaborative Filtering (MF-CF)* [37] and *Spatial-Temporal Neighboring*. The *MAE* of these three missing value estimation approaches is shown in Fig. 12c, d. It is clear that the *MF-CF* approach is much worse than the other two. In the Beijing road network, the *spatial–temporal* approach has a better performance, while in the Shanghai road network, the *cosine similarity* approach is better. The best missing value estimation method of each day has a *MAE* around 1. It means that in a travel of an hour, our speed profile has a travel distance difference about 3 km, which is quite acceptable because different drivers have different driving behavior.

Table 3 Size of speed profiles under different granularities

	1 day (MB)	1 h (MB)	30 min (MB)	15 min (MB)	5 min (MB)
Beijing	4.9	68	135	369	861
Shanghai	4.1	56	112	223	718

7.4.3 Speed profile compression

We compare three compression algorithms on Beijing Map 2015.4.1 in this test: *Sliding Window (SW)*, *Top-Down (TD)* and *Bottom-Up (BU)*. The compression result is shown in Fig. 13. We compare the error *MAE*, compression time and the storage size of each algorithm under error threshold ε of 0.1, 0.5, 1, 2 and 5. As shown in Fig. 13a, the *MAE* of the three algorithms is nearly the same, while the *Bottom-up* is always slightly better than the other two, and it is almost as good as the original one. As expected, the accuracy becomes worse as the compression error threshold ε grows. When it comes to the construction time and space consumption shown in Fig. 13b, c, the *sliding window* algorithm is the fastest to compute and its compression rate is not bad. The *top-down* algorithm is slow and has the worst takes the largest space. The *bottom-up* algorithm takes the longest time to compute, but its compression is the best. In fact, it only takes 18% of original space. So if the compression time is not a problem, we can use the bottom-up algorithm to compress. Otherwise, the sliding window algorithm is a better choice (Table 3).

8 Conclusion

In this paper, we have proposed a route scheduling system that can answer the *MORT* query using the historical trajectories. Our system has an offline speed profile generation component and an online query answering component. On the one hand, different from the previous works that apply their algorithms on synthetic speed profile, our system uses the offline component to generate an accurate and space-efficient speed profile from real-life trajectory. Such an approach involves map matching, speed data collection, missing value estimation and compression. It is cheaper than the traffic monitoring system, and it can cover a larger range of the road network. On the other hand, the online query answering component solves a new route scheduling problem called *MORT* query that aims to minimize on-road time on road networking with parking facilities. *MORT* query further generalizes the path planning problem studied before on road network from allowing the traveler to choose the optimal departure time to minimize on-road travel time that allows multiple stops at parking vertices. From theoretical point of view, *MORT* is the most general type of time-dependent route scheduling problem, which covers all previous problems in terms of both problem formulation and also algorithms. From practical point of view, *MORT* query is useful in many applications, to name a few, minimizing fuel consumption for trucks and advising people to stop and do other things to avoid getting stuck in heavy traffic. From algorithm design and database query processing points of view, *MORT* queries are significantly more complex than the other time-dependent shortest/fastest path queries. We have proposed two algorithms to do *MORT* route scheduling. The *Basic MORT Algorithm* computes a *minimum cost function* directly and takes $O(T|V|\log|V| + T^2|E|)$ time. The *Incremental MORT Algorithm* reduces the time complexity by computing the *minimum cost function* incrementally and takes $O(L(|V|\log|V| + |E|))$ time. An approximate approach α-*MORT* further speeds up the query answering by allowing a guaranteed error bound. Our extensive studies in real-life road networks and trajectories have confirmed that our system could generate accurate and space-saving speed profiles and find minimal on-road time routes more efficiently.

Acknowledgements This research is partially supported by Natural Science Foundation of China (Grant Nos. 61232006, 61502324 and 61532018) and the Australian Research Council (LP130100164 and DP170101172).

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
2. Goldberg, A.V., Harrelson, C.: Computing the shortest path: a search meets graph theory. In: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 156–165. Society for Industrial and Applied Mathematics (2005)
3. Wu, L., Xiao, X., Deng, D., Cong, G., Zhu, A.D., Zhou, S.: Shortest path and distance queries on road networks: an experimental evaluation. Proc. VLDB Endow. **5**(5), 406–417 (2012)
4. Kanoulas, E., Du, Y., Xia, T., Zhang, D.: Finding fastest paths on a road network with speed patterns. In: Proceedings of the 22nd International Conference on Data Engineering, ICDE'06, p. 10. IEEE (2006)
5. Ding, B., Yu, J.X., Qin, L.: Finding time-dependent shortest paths over large graphs. In: Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology, pp. 205–216. ACM (2008)
6. Chabini, I.: Discrete dynamic shortest path problems in transportation applications: complexity and algorithms with optimal run time. Transp. Res. Rec. J. Transp. Res. Board **1645**, 170–175 (1998)
7. Orda, A., Rom, R.: Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. J. ACM (JACM) **37**(3), 607–625 (1990)
8. Demiryurek, U., Banaei-Kashani, F., Shahabi, C., Ranganathan, A.: Online computation of fastest path in time-dependent spatial networks. In: Pfoser, D., et al. (eds.) Advances in spatial and temporal databases, pp. 92–111. Springer, Berlin (2011)
9. Cai, X., Kloks, T., Wong, C.: Time-varying shortest path problems with constraints. Networks **29**(3), 141–150 (1997)
10. Dreyfus, S.E.: An appraisal of some shortest-path algorithms. Oper. Res. **17**(3), 395–412 (1969)
11. Demiryurek, U., Pan, B., Banaei-Kashani, F., Shahabi, C.: Towards modeling the traffic data on road networks. In: Proceedings of the Second International Workshop on Computational Transportation Science, pp. 13–18. ACM (2009)
12. Zheng, B., Su, H., Hua, W., Zheng, K., Zhou, X., Li, G.: Efficient clue-based route search on road networks. IEEE Trans. Knowl. Data Eng. **29**, 1846 (2017)
13. Li, L., Hua, W., Du, X., Zhou, X.: Minimal on-road time route scheduling on time-dependent graphs. Proc. VLDB Endow. **10**(11), 1274–1285 (2017)
14. Cooke, K.L., Halsey, E.: The shortest route through a network with time-dependent internodal transit times. J. Math. Anal. Appl. **14**(3), 493–498 (1966)
15. Geisberger, R.: Contraction of timetable networks with realistic transfers. In: Festa, P. (ed.) Experimental algorithms, pp. 71–82. Springer, Berlin (2010)
16. Wu, H., Cheng, J., Huang, S., Ke, Y., Lu, Y., Xu, Y.: Path problems in temporal graphs. Proc. VLDB Endow. **7**(9), 721–732 (2014)
17. Wang, S., Lin, W., Yang, Y., Xiao, X., Zhou, S.: Efficient route planning on public transportation networks: a labelling approach. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 967–982. ACM (2015)
18. Halpern, J.: Shortest route with time dependent length of edges and limited delay possibilities in nodes. Z. Oper. Res. **21**(3), 117–124 (1977)
19. Orda, A., Rom, R.: Minimum weight paths in time-dependent networks. Networks **21**(3), 295–319 (1991)
20. Foschini, L., Hershberger, J., Suri, S.: On the complexity of time-dependent shortest paths. Algorithmica **68**(4), 1075–1097 (2014)
21. Cai, X., Kloks, T., Wong, C.: Shortest path problems with time constraints. In: International Symposium on Mathematical Foundations of Computer Science, pp. 255–266. Springer (1996)
22. Batz, G.V., Delling, D., Sanders, P., Vetter, C.: Time-dependent contraction hierarchies. In: Proceedings of the Meeting on Algorithm Engineering and Experiments, pp. 97–105. Society for Industrial and Applied Mathematics (2009)

23. Delling, D.: Time-dependent sharc-routing. Algorithmica **60**(1), 60–94 (2011)
24. Li, L., Zhou, X., Zheng, K.: Finding least on-road travel time on road network. In: Australasian Database Conference, pp. 137–149. Springer (2016)
25. Yang, Y., Gao, H., Yu, J.X., Li, J.: Finding the cost-optimal path with time constraint over time-dependent graphs. Proc. VLDB Endow. **7**(9), 673–684 (2014)
26. Adler, J.D., Mirchandani, P.B., Xue, G., Xia, M.: The electric vehicle shortest-walk problem with battery exchanges. Netw. Spat. Econ. **16**(1), 155–173 (2016)
27. Ichimori, T., Ishii, H., Nishida, T.: Routing a vehicle with the limitation of fuel. J. Oper. Res. Soc. Jpn. **24**(3), 277–281 (1981)
28. Xiao, Y., Thulasiraman, K., Xue, G., Jüttner, A.: The constrained shortest path problem: algorithmic approaches and an algebraic study with generalization. AKCE Int. J. Graphs Comb. **2**(2), 63–86 (2005)
29. Wang, S., Xiao, X., Yang, Y., Lin, W.: Effective indexing for approximate constrained shortest path queries on large road networks. Proc. VLDB Endow. **10**(2), 61–72 (2016)
30. Blokh, D., Gutin, G.: An approximate algorithm for combinatorial optimization problems with two parameters. Australas. J. Comb. **14**, 157–164 (1996)
31. Juttner, A., Szviatovski, B., Mécs, I., Rajkó, Z.: Lagrange relaxation based method for the QoS routing problem. In: Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2001, vol. 2, pp. 859–868. IEEE (2001)
32. Tong, Y., Wang, L., Zhou, Z., Ding, B., Chen, L., Ye, J., Xu, K.: Flexible online task assignment in real-time spatial data. Proc. VLDB Endow. **10**(11), 1334–1345 (2017)
33. Tong, Y., Chen, Y., Zhou, Z., Chen, L., Wang, J., Yang, Q., Ye, J., Lv, W.: The simpler the better: a unified approach to predicting original taxi demands based on large-scale online platforms. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1653–1662. ACM (2017)
34. Dai, J., Yang, B., Guo, C., Jensen, C.S., Hu, J.: Path cost distribution estimation using trajectory data. PVLDB **10**(3), 85–96 (2016)
35. Bakalov, P., Hoel, E., Heng, W.-L.: Time dependent transportation network models. In: 2015 IEEE 31st International Conference on Data Engineering (ICDE), pp. 1364–1375. IEEE (2015)
36. Yang, B., Guo, C., Jensen, C.S.: Travel cost inference from sparse, spatio temporally correlated time series using Markov models. Proc. VLDB Endow. **6**(9), 769–780 (2013)
37. Shang, J., Zheng, Y., Tong, W., Chang, E., Yu, Y.: Inferring gas consumption and pollution emission of vehicles throughout a city. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1027–1036. ACM (2014)
38. Xin, X., Lu, C., Wang, Y., Huang, H.: Forecasting collector road speeds under high percentage of missing data. In: AAAI, pp. 1917–1923 (2015)
39. Asif, M.T., Mitrovic, N., Garg, L., Dauwels, J., Jaillet, P.: Low-dimensional models for missing data imputation in road networks. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 3527–3531. IEEE (2013)
40. Shan, Z., Zhao, D., Xia, Y.: Urban road traffic speed estimation for missing probe vehicle data based on multiple linear regression model. In: 16th International IEEE Conference on Intelligent Transportation Systems-(ITSC), pp. 118–123. IEEE (2013)
41. Widhalm, P., Piff, M., Brändle, N., Koller, H., Reinthaler, M.: Robust road link speed estimates for sparse or missing probe vehicle data. In: 15th International IEEE Conference on Intelligent Transportation Systems (ITSC), pp. 1693–1697. IEEE (2012)
42. Guo, C., Jensen, C.S., Yang, B.: Towards total traffic awareness. SIGMOD Rec. **43**(3), 18–23 (2014)
43. Guo, C., Yang, B., Andersen, O., Jensen, C.S., Torp, K.: Ecomark 2.0: empowering eco-routing with vehicular environmental models and actual vehicle fuel consumption data. GeoInformatica **19**(3), 567–599 (2015)
44. Idé, T., Sugiyama, M.: Trajectory regression on road networks. In: AAAI (2011)
45. Zheng, J., Ni, L.M.: Time-dependent trajectory regression on road networks via multi-task learning. In: AAAI (2013)
46. Yang, B., Kaul, M., Jensen, C.S.: Using incomplete information for complete weight annotation of road networks. IEEE Trans. Knowl. Data Eng. **26**(5), 1267–1279 (2014)
47. Zhang, J., Zheng, Y., Qi, D.: Deep spatio-temporal residual networks for citywide crowd flows prediction. In: AAAI, pp. 1655–1661 (2017)
48. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM (JACM) **34**(3), 596–615 (1987)
49. Lou, Y., Zhang, C., Zheng, Y., Xie, X., Wang, W., Huang, Y.: Map-matching for low-sampling-rate GPS trajectories. In: Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 352–361. ACM (2009)
50. Yuan, J., Zheng, Y., Zhang, C., Xie, X., Sun, G.-Z.: An interactive-voting based map matching algorithm. In: Proceedings of the 2010 Eleventh International Conference on Mobile Data Management, pp. 43–52. IEEE Computer Society (2010)
51. Quddus, M.A., Ochieng, W.Y., Noland, R.B.: Current map-matching algorithms for transport applications: state-of-the art and future research directions. Transp. Res. Part C Emerg. Technol. **15**(5), 312–328 (2007)
52. Cox, D.R.: The regression analysis of binary sequences. J. R. Stat. Soc. Ser. B (Methodol.) **1**, 215–242 (1958)
53. Seal, H.L.: The Historical Development of the Gauss Linear Model. Yale University, New Haven (1968)
54. Shatkay, H., Zdonik, S.B.: Approximate queries and representations for large data sequences. In: Proceedings of the Twelfth International Conference on Data Engineering, pp. 536–545. IEEE (1996)
55. Keogh, E., Chu, S., Hart, D., Pazzani, M.: Segmenting time series: a survey and novel approach. Data Min. Time Ser. Databases **57**, 1–22 (2004)
56. Esling, P., Agon, C.: Time-series data mining. ACM Comput. Surv. (CSUR) **45**(1), 12 (2012)
57. Li, C.-S., Yu, P.S., Castelli, V.: Malm: A framework for mining sequence database at multiple abstraction levels. In: Proceedings of the Seventh International Conference on Information and Knowledge Management, pp. 267–272. ACM (1998)
58. Keogh, E J., Pazzani, M J.: An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. KDD **98**, 239–243 (1998)