

Minimal On-Road Time Route Scheduling on Time-Dependent Graphs

Lei Li*, Wen Hua*, Xingzhong Du*, Xiaofang Zhou*†

*School of Information Technology and Electrical Engineering, The University of Queensland, Australia

†School of Computer Science and Technology, Soochow University, China

{l.li3, w.hua, x.du, uqxzhou}@uq.edu.au

ABSTRACT

On time-dependent graphs, fastest path query is an important problem and has been well studied. It focuses on minimizing the total travel time (waiting time + on-road time) but does not allow waiting on any intermediate vertex if the *FIFO* property is applied. However, in practice, waiting on a vertex can reduce the time spent on the road (for example, resuming traveling after a traffic jam). In this paper, we study how to find a path with the minimal on-road time on time-dependent graphs by allowing waiting on some predefined parking vertices. The existing works are based on the following fact: the arrival time of a vertex v is determined by the arrival time of its in-neighbor u , which does not hold in our scenario since we also consider the waiting time on u if u allows waiting. Thus, determining the waiting time on each parking vertex to achieve the minimal on-road time becomes a big challenge, which further breaks *FIFO* property. To cope with this challenging problem, we propose two efficient algorithms using *minimum on-road travel cost function* to answer the query. The evaluations on multiple real-world time-dependent graphs show that the proposed algorithms are more accurate and efficient than the extensions of existing algorithms. In addition, the results further indicate, if the parking facilities are enabled in the route scheduling algorithms, the on-road time will reduce significantly compared to the fastest path algorithms.

1. INTRODUCTION

With the prevalence of GPS enabled devices and wireless network, navigation systems have been widely adopted by public transportation, logistics, private vehicles and a broad range of location-based services. Essentially, it is the path planning algorithm that plays the vital role in those navigation systems, which helps people travel more smartly and more predictively. In the past decades, different path planning algorithms are proposed for various application scenarios and requirements. For example, shortest path algorithms [1, 2, 3, 4] find a path with the minimal distance between

origin and destination, while fastest path algorithms return a path with the least total travel time given a static traffic condition [5]. If a user is allowed to depart from any time during a certain period, another set of fastest path algorithms can be used [6, 7, 8, 9, 10, 11] to find the optimal departure time with the least total travel time. Moreover, path planning algorithms for earliest arrival and latest departure [12, 13] are also important in transportation.

The common optimization goal of the above path planning algorithms is the total travel time, which is the difference between departure time and arrival time, and is made up of 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 problem makes 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 [5]. 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 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 in order to avoid predictable traffic jam. To better understand how waiting can shorten the on-road time when traveling, consider the road network with five vertices shown in Figure 1. Three of them are ordinary vertices, and two of them are parking vertices that allow waiting. The traveling cost functions are shown in Figure 1(b)-(f). 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 advan-

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.

Proceedings of the VLDB Endowment, Vol. 10, No. 11
Copyright 2017 VLDB Endowment 2150-8097/17/07.

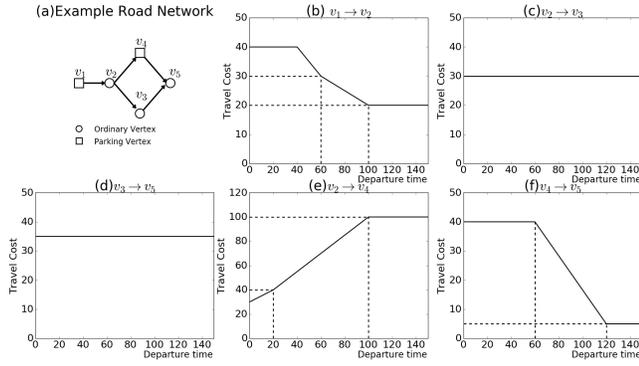


Figure 1: A road networking with parking vertices (a) and the corresponding time-dependent weight for each edge over time domain (0 - 150) (b) - (f)

tages of these parking vertices, we can obtain a route that has shorter on-road travel time. More application scenarios are explained in Section 4.4 after the algorithm is fully described.

In this work, we model a road network as a time-dependent graph, whereas 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. 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 path query (MORT)*, aims to find a path 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 to 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, 2, 3, 4, 14] only works with static edge weights. Thus, it cannot handle the time-dependent costs. Secondly, the *single starting-time fastest path(SSFP)* algorithm [5] 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 [6, 7] 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* path 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).

In this paper, we propose two algorithms to find the minimal on-road travel route. Both of them 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 time. 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 minimum staying 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*.

In summary, our contributions are listed as follows:

- We identify a general form of time-dependent route scheduling problem, called *MORT*, to make 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. 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* runs *MORT* search for each vertex 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.

- We evaluate the effectiveness and efficiency of our *MORT* algorithms with extensive experiments on road network and small world graphs, measuring both the reduction of the minimal on-road time and the algorithm running time.

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 Section 3. Section 4 presents the two *MORT* algorithms with correctness and complexity analysis. An empirical study is shown in Section 5. Our conclusions can be found in Section 6.

2. RELATED WORK

In this section, we review the previous works on modeling time-dependent road network and position our work by discussing the difference from the fastest 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. [15] proved that these queries could be solved with a modified 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 [16, 12, 13], 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 [5] 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 [17] and [18]. 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 a 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 [5]. 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. But once departing, no waiting is allowed along the path. The difference between *ISFP* and *MORT* is illustrated in Figure 2. 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 [19] that the theoretical lower-bound of *ISFP* is $\Omega(T(|V| \log |V| + |E|))$ [19], where T is the average number turning points in the

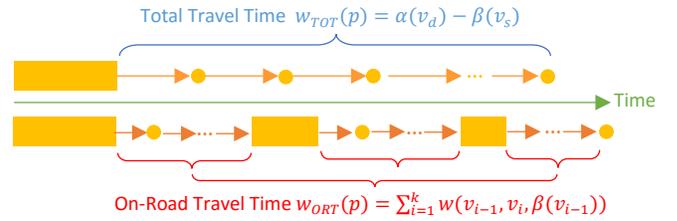


Figure 2: Comparison between *total travel time* and *on-road travel time*. Thick bar: Waiting time on a parking vertex; Circle: No waiting on the vertex; Arrow: Travel time from one vertex to another

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* [8] and [20, 11] 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 . [6] 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. [7] 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 build different kinds of indexes to speed up fastest path query, such as *time-dependent CH* [21] and *time-dependent SHARC* [22].

Although *ISFP* is different from *MORT*, we can adopt it as our baseline algorithm by invoking the algorithms in [6, 7] recursively to get an approximate result. [23, 24] take waiting on intermediate vertices into consideration in their problems. But they allow waiting on any vertex, which does not make sense in real life. In fact, [23] 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 [24], 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).

3. PROBLEM DEFINITION

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 there is no edge from u to v .

A path from u to v in G can be represented as $p = \langle v_0, v_1, \dots, v_k \rangle$, 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 path from u to v in G .

Now let us differentiate two different types of cost for a path: the *total travel time* $w_{TOT}(p) = \alpha(v_k) - \beta(v_0)$ and the *on-road travel time* $w_{ORT}(p) = \sum_{i=1}^k w(v_{i-1}, v_i, \beta(v_{i-1}))$. Although $w_{ORT}(p)$ 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 to 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 depart 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 departs earlier will always get to the destination earlier (known as the *FIFO* property [5]). 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 [6, 7]. 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 $v \in V - V'$. 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 path from v_s to v_d , represented as $p = \langle v_0, v_1, \dots, v_k \rangle$, 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 paths meeting the conditions (1), (2), (3) and (4).

Condition (1) means that p is a path 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 path meeting these four conditions, the cost to travel from v_s to v_d is defined as ∞ . Condition 5 requires the path 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 [6, 7] is a special case of the *MORT* query where parking node set $V' = \{v_s\}$.

4. ALGORITHM

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* path 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 sub-intervals, 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 path expanding algorithms are able to find the *MORT* time, its result is not a route schedule, which is the expected output of *MORT* problem. To address that, *path retrieval* is introduced to generate the final results. Considering scalability is important for route scheduling, we present the correctness and complexity analysis of the proposed method 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. **Path Expansion** finds the path 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 path

Table 1: Important Notations

Notation	Description
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 $g_{f,i}(t)$
$C'_i(t)$	$\min(C_i(t), g_{f,i}(t))$

expansion part. The full details of the path expansion which are the major contributions in this work will be presented in Section 4.2 and 4.3, respectively. We further explain how to apply our algorithms to different scenarios in Section 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 an 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, are 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 [24], 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 *path expansion*. Notice that when t is given, we use $w(v_i, v_j, t)$ to represent the time cost of travelling from v_i to v_j at time t . The speed profile is then instantiated as $\{(t_0, w(v_i, v_j, t_0)), \dots, (t_k, w(v_i, v_j, t_k))\}$, and the intermediate values between points are computed linearly. Figure 1(b)-(f) illustrate 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, respectively. 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 query the road network in Figure 1 with $Q_{MORT}(v_1, v_5, 0, 30, 130)$ 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 [24]. Accordingly, the output of path 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 path 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.

$C_i(t)$ 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 $S_i = \{(t_0, C_i(t_0)), \dots, (t_k, C_i(t_k))\}$. 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 path 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.

If t is a user-specified arrival time, we can traverse the vertices back from v_d at time t . In this backward traversal, 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 $t'_i \leq t_i$. 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.

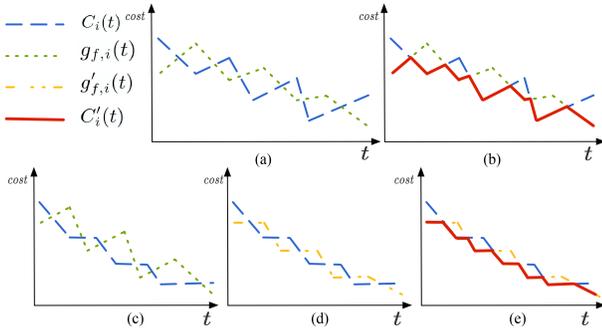


Figure 3: Minimum Cost Function Update

(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

4.2 Basic MORT Algorithm

The *Path 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 in 4.2.1, then present path expansion in 4.2.2. Correctness and complexity are proved in 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 Figure 3(a)-(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 3(c) 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 Figure 3(d), and then compute $C'_i(t)$ in Figure 3(e). 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 $\min(C_i(t_b), g_{f,i}(t_b)) = C_i(t_b)$,

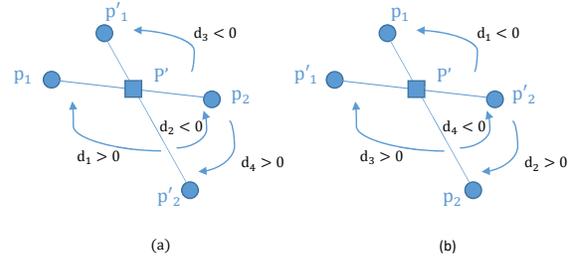


Figure 4: Line Segment Intersection

$g_{f,i}(t_a) - g_{f,i}(t_b) - C_i(t_b)$, non-increasing holds. The remaining two situations are similar. \square

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 $g_{f,i}(t'') = C_f(t') + w(v_f, v_i, t')$.

The details of *MCFU* is 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. Line 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 (Line 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 segment in $C_i(t)$ and $g'_{f,i}(t)$ (Line 9-10), and their corresponding end points (p_1, p_2) and (p'_1, p'_2) (Line 12-13). Line 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 Figure 4. 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. Given 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 $O(T)$.

4.2.2 Basic Path Expansion Algorithm

Path 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. Line 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 (Line

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   $t_2 \leftarrow S_f[0], t'_2 \leftarrow S_f[1]$  //  $S_f$ : 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  $(p_k - p_i) \times (p_j - p_i)$ 
```

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 (Line 9-10).

4.2.3 Correctness

THEOREM 3. *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 path 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. \square

4.2.4 Complexity Analysis

As mentioned previously, the time complexity of the ATU algorithm is $O(|V| \log |V| + |E|)$. As for the Path Expansion algorithm, we use Fibonacci Heap [25] to implement

Algorithm 2: Path Expansion Algorithm

Input: $G(V, E)$, $Q_{MORT}(v_s, v_d, t_{s1}, t_{s2}, t_d)$

Output: $R_{p^*,d}$

```
1 begin
2   for  $v_i \in V$  do
3      $C_i(v_i.t_{EA}) \leftarrow v_i.t_{EA} - t_{s1}$ 
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  $\min(C_d(t))$ 
```

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 Path Expansion to build the minimum cost function for each vertex v_i in its $T_i = [v_i.t_{EA}, v_i.t_{LD}]$ sub-interval by sub-interval incrementally, which could reduce unnecessary computations.

4.3.1 Incremental Path 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 sub-interval $I'_i = [v_i.t_{EA}, \tau'_i] \subseteq T_i$ 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 sub-interval 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 $[\tau'_j, \tau_j^2]$. 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 sub-interval to refine, and *Minimum Cost Function Update*.

Algorithm 3: Incremental Path 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    $C_s(t_s) \leftarrow 0$ ,  $C_s(v_s.t_{LD}) \leftarrow 0$ ,  $\tau_s \leftarrow t_s$ 
3   for  $v_i \in V \setminus \{v_s\}$  do
4      $C_i(v_i.t_{EA}) = v_i.t_{EA} - t_s$ ,  $\tau_i \leftarrow v_i.t_{EA}$ 
5   Let  $Q$  be a priority queue initially containing pairs
    $(\tau_i, C_i(t))$ , ordered by  $\tau_i$  in ascending order
6   while  $|Q| \geq 2$  do
7      $(\tau_i, C_i(t)) \leftarrow Q.pop()$ 
8      $(\tau_k, C_k(t)) \leftarrow Q.top()$ 
9      $\tau'_i \leftarrow \tau_k + \min\{w(v_f, v_i, \tau_k) | v_f \text{ is } v_i\text{'s in-neighbor}\}$ 
10    for  $v_j$  is  $v_i$ 's out-neighbor do
11      if  $v_i \in v'$  then
12         $C'_j(t') \leftarrow C_i(t') + w(v_i, v_j, t')$ 
13         $t' \leftarrow t + w(v_i, v_j, t)$ ,  $t'' \leftarrow t' + v_i.t_{min}$ 
14      else
15         $C'_j(t') \leftarrow C_i(t) + w(v_i, v_j, t)$ 
16         $t' \leftarrow t + w(v_i, v_j, t)$ 
17         $t \in [\tau_i, \tau'_i]$ 
18        if  $v_j \in V'$  then
19           $C'_j(t) \leftarrow Non - Increase(C'_j(t))$ 
20           $\tau_j^1 = \tau_i + w(v_i, v_j, \tau_i)$ 
21           $\tau_j^2 = \tau'_i + w(v_i, v_j, \tau'_i)$ 
22           $C_j(t) \leftarrow \min(C_j(t), C'_j(t), t \in [\tau_j^1, \tau_j^2])$ 
23           $Q.update(\tau_j, C_j(t))$ 
24         $\tau_i \leftarrow \tau'_i$ 
25        if  $v_i = v_d$  and  $\tau_i \geq t_d$  then
26          return  $\min(C_i(t))$ 
27        else if  $\tau_i < v_i.t_{LD}$  then
28           $Q.insert((\tau_i, C_i(t)))$ 
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 $v_i.t_{EA}$ (Line 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 (Line 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 (Line 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 (Line 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 Section 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 $\langle (\tau_1 = 0, C_1(t)), (\tau_2 = 40, C_2(t)), (\tau_3 = 70, C_3(t)), (\tau_4 = 95, C_4(t)), (\tau_5 = 105, C_5(t)) \rangle$. $C_0(t)$ 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 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 $\langle (\tau_2 = 40, C_2(t)), (\tau_3 = 70, C_3(t)), (\tau_4 = 95, C_4(t)), (\tau_5 = 105, C_5(t)) \rangle$. 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 6, we first prove the minimum cost function is correctly computed. Lemma 4 proves Line 7-9 is correct. Lemma 5 proves Line 10-23 is correct.

LEMMA 4. *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 $[\tau_i, \tau_i']$. \square

LEMMA 5. *$C_i(t)$, 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 $[\tau_i, \tau_i']$. \square

THEOREM 6. *Algorithm 3 finds the MORT.*

PROOF. Lemma 5 guarantees each $C_i(t)$ is fully updated, and Lemma 4 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. \square

4.3.4 Complexity Analysis

The *ATI* takes $O(|V| \log |V| + |E|)$ time. The initialization phase (Line 2-5) takes $O(V)$ time. We use *Fibonacci Heap* [25] 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 $O(\sum_{i=0}^{|V|} L_i |V| \log |V|) = O(L(|V| \log |V|))$ 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, $\sum_{i=0}^{|V|} |E_i^{in}| = |E|$. So the total time complexity is $O(\sum_{i=0}^{|V|} L_i (\log |V| + |E_i^{in}|)) = O(L(|V| \log |V| + |E|))$.

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.

First, 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.

Second, consider 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. The traveling time between these stations roughly equals to the driver's maximum driving time. Therefore, the force waiting is included in the computation and the minimum rest time is guaranteed for safe driving.

Finally, suppose a traveler is planning a journey from one city to another in several weeks time and wants to visit national parks along the 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 on the same layer. In an extreme case when the traveler wants to visit every park, the graph should be organized as a linear

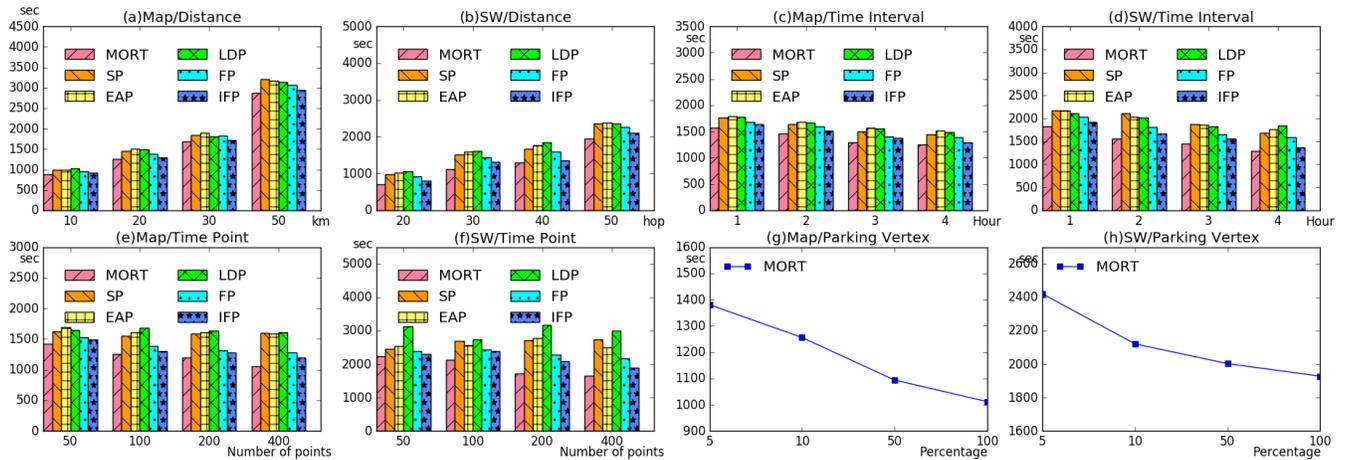


Figure 5: Results of Minimal On-Road Time

line. The edges only exist between the vertices in neighboring layers. 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 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. EXPERIMENTS

In this section, we present the results of a comprehensive performance study on one real-world road networks and a small-world graph with different speed profiles, to demonstrate the effectiveness and efficiency of our algorithms.

5.1 Experiment Setup

We test our method on two types of graph. The first one is a real world road network of Beijing, which consists of 302,364 intersections and 387,588 roads (60MB). The second one is generated from *Watts and Strogatz Small World Model* [26] and consists of 100K vertices and 400K edges (36MB). Such a graph pattern can be found in many real-life networks like social network, computer communication network, phone call network and brain neuron network.

We generate four speed profiles for each graph. Each of them has 50, 100, 200 and 400 random turning points in a total number of 86400 time points (second number of one day). The values are randomly chosen from 5 to 100. The sizes of them are around 170MB, 360MB, 670MB and 1.3GB.

We test the algorithms under four variations. The first one is the distance of two vertices for the map and the hop number for the other two graphs. The second one varies the starting time interval size from 1 hour, 2 hours, 3 hours to 4 hours. 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%).

We ran all the experiments on a Dell R720 PowerEdge Rack Mount Server which has two Xeon E5-2690 2.90GHz CPUs, 192GB memory and runs Ubuntu Server 14.04 LTS operating system. All the programs run in single thread.

5.2 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*) [7]. 4) *IFP* (*Iterative Fastest Path*) which uses the *FP* (*Fastest Path*) algorithm iteratively to get the approximate minimal on-road time path, as described in Section 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 10km, 20km, 30km, and 50km in Beijing map, and hop number of 20, 30, 40, 50 in the other two graphs. The starting time interval is set to be 4 hours. 10% of the vertices are selected as parking vertices. We use 100-point speed profile in this test. The results on three graphs are shown in Figure 5(a)-(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. In addition, the time on the map is always longer than small-world graph because it actually has a larger number of hop numbers.

The second test varies the length of starting time interval from 1 hour to 4 hours. The distance/hop number is set to be 20, speed profile is 100 and parking vertex is 10%. Figure 5(c)-(d) show the results. As the length of the time

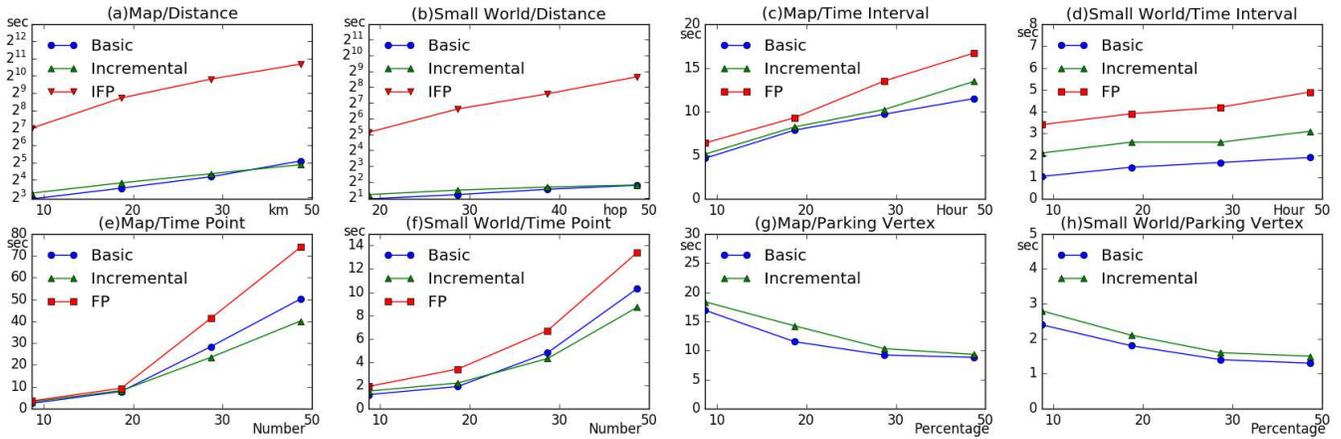


Figure 6: Algorithm Running Time

interval grows, more possible starting time emerge, so the on-road time of *FP* and *IFP* decrease. As for *MORT*, it also decreases because it has a longer time to wait for a faster path 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, whose turning point numbers are 50, 100, 200 and 400. The distance/hop number is also 20, parking vertex is 10% and the starting time interval is 4 hours. We can see from Figure 5(e)-(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 paths 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, speed profile is 100 and time interval is 4 hours. Figure 5(g)-(h) only show 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 faster speed.

5.3 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, Figure 6(a)-(b) show the results under different distances. As the distance/hop number grows, the numbers of the visited 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 Figure 6(c)-(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 Figure 6(e)-(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 Figure 6(c)-(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 Figure 6(g)-(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 not 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.

6. CONCLUSION

In this paper, we have studied a new route scheduling problem called *MORT* query that aims to minimize on-road time in time-dependent graphs with parking vertices. *MORT* query further generalizes the path planning problem studied before in time-dependent graphs 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 both in terms of 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 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. Our extensive studies in road network and small-world graph have confirmed that our algorithms could find minimal on-road time paths more efficiently.

7. ACKNOWLEDGMENT

This research is partially supported by Natural Science Foundation of China (Grant No. 61232006) and the Australian Research Council (Grant No. LP130100164).

8. REFERENCES

- [1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
- [2] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2005, pp. 156–165.
- [3] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou, "Shortest path and distance queries on road networks: An experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 5, no. 5, pp. 406–417, 2012.
- [4] S. Wang, X. Xiao, Y. Yang, and W. Lin, "Effective indexing for approximate constrained shortest path queries on large road networks," *PVLDB*, vol. 10, no. 2, pp. 61–72, 2016.
- [5] S. E. Dreyfus, "An appraisal of some shortest-path algorithms," *Operations research*, vol. 17, no. 3, pp. 395–412, 1969.
- [6] E. Kanoulas, Y. Du, T. Xia, and D. Zhang, "Finding fastest paths on a road network with speed patterns," in *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE, 2006, pp. 10–10.
- [7] B. Ding, J. X. Yu, and L. Qin, "Finding time-dependent shortest paths over large graphs," in *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. ACM, 2008, pp. 205–216.
- [8] I. Chabini, "Discrete dynamic shortest path problems in transportation applications: Complexity and algorithms with optimal run time," *Transportation Research Record: Journal of the Transportation Research Board*, no. 1645, pp. 170–175, 1998.
- [9] A. Orda and R. Rom, "Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length," *Journal of the ACM (JACM)*, vol. 37, no. 3, pp. 607–625, 1990.
- [10] U. Demiryurek, F. Banaei-Kashani, C. Shahabi, and A. Ranganathan, "Online computation of fastest path in time-dependent spatial networks," in *Advances in spatial and temporal databases*. Springer, 2011, pp. 92–111.
- [11] X. Cai, T. Kloks, and C.-K. Wong, "Time-varying shortest path problems with constraints," *Networks*, vol. 29, no. 3, pp. 141–150, 1997.
- [12] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 9, pp. 721–732, 2014.
- [13] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou, "Efficient route planning on public transportation networks: A labelling approach," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 967–982.
- [14] B. Zheng, H. Su, W. Hua, K. Zheng, X. Zhou, and G. Li, "Efficient clue-based route search on road networks," *IEEE Transactions on Knowledge and Data Engineering*, 2017.
- [15] K. L. Cooke and E. Halsey, "The shortest route through a network with time-dependent internodal transit times," *Journal of mathematical analysis and applications*, vol. 14, no. 3, pp. 493–498, 1966.
- [16] R. Geisberger, "Contraction of timetable networks with realistic transfers," in *Experimental Algorithms*. Springer, 2010, pp. 71–82.
- [17] J. Halpern, "Shortest route with time dependent length of edges and limited delay possibilities in nodes," *Zeitschrift fuer operations research*, vol. 21, no. 3, pp. 117–124, 1977.
- [18] A. Orda and R. Rom, "Minimum weight paths in time-dependent networks," *Networks*, vol. 21, no. 3, pp. 295–319, 1991.
- [19] L. Foschini, J. Hershberger, and S. Suri, "On the complexity of time-dependent shortest paths," *Algorithmica*, vol. 68, no. 4, pp. 1075–1097, 2014.
- [20] X. Cai, T. Kloks, and C. Wong, "Shortest path problems with time constraints," in *International Symposium on Mathematical Foundations of Computer Science*. Springer, 1996, pp. 255–266.
- [21] G. V. Batz, D. Delling, P. Sanders, and C. Vetter, "Time-dependent contraction hierarchies," in *Proceedings of the Meeting on Algorithm Engineering & Experiments*. Society for Industrial and Applied Mathematics, 2009, pp. 97–105.
- [22] D. Delling, "Time-dependent sharc-routing," *Algorithmica*, vol. 60, no. 1, pp. 60–94, 2011.
- [23] L. Li, X. Zhou, and K. Zheng, "Finding least on-road travel time on road network," in *Australasian Database Conference*. Springer, 2016, pp. 137–149.
- [24] Y. Yang, H. Gao, J. X. Yu, and J. Li, "Finding the cost-optimal path with time constraint over time-dependent graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 9, pp. 673–684, 2014.
- [25] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.
- [26] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-world networks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.