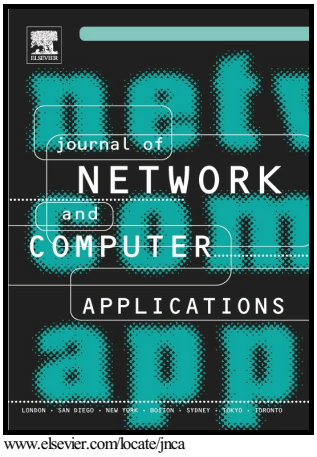
Discovering Time-dependent Shortest Path on Traffic Graph for Drivers towards Green Driving

Yunchuan Sun, Xinpei Yu, Rongfang Bie, Houbing Song



www.elsevier.com/locate/jnca

PII: S1084-8045(16)00049-7
DOI: http://dx.doi.org/10.1016/j.jnca.2015.10.018
Reference: YJNCA1555

To appear in: *Journal of Network and Computer Applications*

Received date: 6 February 2015
Revised date: 6 October 2015
Accepted date: 13 October 2015

Cite this article as: Yunchuan Sun, Xinpei Yu, Rongfang Bie and Houbing Song, Discovering Time-dependent Shortest Path on Traffic Graph for Drivers towards Green Driving, *Journal of Network and Computer Applications*, http://dx.doi.org/10.1016/j.jnca.2015.10.018

Discovering Time-dependent Shortest Path on Traffic Graph for Drivers towards Green Driving

Yunchuan Sun[1], Xinpei Yu[2], Rongfang Bie[2], Houbing Song[3]

[1] Business School, Beijing Normal University, Beijing, 10085 China

[2]Information Science and Technology,Beijing Normal University, Beijing, 10085 China

[3]Department of Electrical and Computer Engineering, West Virginia University, Montgomery, WV 25136 USA

Email: {yunch,rfbie}@bnu.edu.cn,xinpyu@gmail.com, Houbing.Song@mail.wvu.edu

***Abstract*—Green transportation technologies that reduce fuel consumption, idling, and vehicle miles traveled while reducing acute congestion could play a significant role in reducing greenhouse gas emissions, particularly in major cities, around ports and freight hubs, and on major roads and corridors. The key to support green transportation is to discover real-time shortest path for drivers. A time-dependent traffic graph, generated from history traffic data can be used to predict the travelling time and to find the shortest path for drivers dynamically. There are two different types of queries on shortest paths: *Query_FiST* for starting at a fixed time and reaching the destination as early as possible and *Query_BeST* for choosing the best starting time to avoid the rush hours. Taking advantage of traffic graph's characteristics of sparsity and hierarchy, we propose two algorithms: Heap-based Bellman-Ford algorithm for *Query_FiST* and Extended Bellman-Ford algorithm for *Query_BeST* respectively. We prove the correctness of the algorithms and discuss their time complexity. A series of experiments are implemented on an open data set of real traffic collected from taxis and the results show that our algorithms outperform all existing algorithms practically.**

***Index Terms*—shortest path problem, optimization, green driving**

I. INTRODUCTION

Our world is facing serious surface transportation challenges, including safety, mobility, and environment. From the point of view of the environmental impacts of transportation, on one hand, vehicle fuel utilization and its tailpipe emissions are the largest human-made source of carbon dioxide, nitrous oxide, and methane, which cause pulmonary diseases and premature deaths. On the other hand, vehicles that are stationary, idling, or traveling at reduced speeds due to congestion emit more than those that are in free flow conditions. Therefore, green transportation technologies that reduce fuel consumption, idling, and vehicle miles traveled while reducing acute congestion could play a significant role in reducing greenhouse gas emissions, particularly in major cities, around ports and freight hubs, and on major roads and corridors. Informed drivers may decide to avoid congested routes, take alternate routes, use public transit, or reschedule their trip, all of which can make their trip more fuel-efficient and eco-friendly. The key is to discover real-time shortest path for drivers. The purpose of this paper is to address this problem.

The advancement in technologies of locating and tracking enables us to obtain real-time road condition information including its location, volume, average speed, and road bumpy status in several ways, such as Intelligent Transportation Systems (ITS), which collect transportation information by various sensors and monitoring devices [1] or smart phones embedded GPS modules and accelerometers [5]. Road condition information could be used to identify the level of current congestion and then to predict future traffic status according to patterns extracted from history traffic data [8][9].

The traffic prediction varies from the mode of transportation: by buses, by private cars or taxis. For buses, the prediction focuses on the arrival time due to the fixed driving scheduling (fixed paths and fixed time to set out). Accurate prediction of the arrival time could help passengers schedule their travels and improve their experience, so several works were proposed recent years [2][3][4]. Compared to buses, private cars or taxis have a different mode: drivers can freely schedule their paths and time to set out. Drivers are more inclined to find the optimal travel path according to real-time road condition. It would be better for them if the intelligent systems could provide the shortest path dynamically in time. A traffic graph containing road network information and future traffic status is indispensable for computing the shortest path dynamically. Using existing traffic prediction algorithms[8][9], the traffic graph can be worked out in advance.

For drivers of cars or taxis, there are two different services or applications we could supply. 1) Imagine an ordinary scene, a driver hopes to start right now and reach the destination as early as possible. Obviously, the time to set out and the destination are fixed for the algorithms. In this case, the query for the shortest path with a fixed starting time can be denoted as *Query_FiST* (*Query* on *Fi*xed *S*tarting *T*ime). 2) Consider there's an express company which uses trucks to deliver goods for customs. A truck may travel less time if it chose the best starting time to avoid the rush hour. The best starting time needs to be calculated according to the traffic graph. The query in this case for the shortest path with the best starting time can be denoted as *Query_BeST* (*Query* on *Be*st *S*tarting *T*ime).

There have been some fundamental researches in last decades [18]. Two most famous algorithms, Dijkstra's algorithm and Bellman-Ford algorithm, have been proposed to solve the shortest path problems in static graphs. Dijkstra's algorithm is essentially a graph search algorithm using greedy method with time complexity of $O(V^2)$. Bellman-Ford algorithm labels nodes with their distances to the source node or their earliest arrival time through iterations of updating the labels on each node. It can be implemented easily and able to support negative edge weights, but it is not efficient for its time complexity of $O(VE)$.

Regarding the dynamic graphs like traffic graphs, many studies aim to discover the shortest paths by improving Dijkstra's algorithm [12][19] and Bellman-Ford algorithm [11]. Other than reducing the complexity of the original algorithms, these optimizations enable the algorithms to handle dynamic graphs. Although the optimizations of Bellman-Ford algorithm are slower than that of Dijkstra's algorithm due to the reason that the number of edges E is much larger than nodes V in most cases, some optimizations of Bellman-Ford algorithms perform really well for some special kinds of dynamic graphs. The motivation of this paper is to develop some excellent algorithms for finding the shortest paths by optimizing the Bellman-Ford algorithm based on the following characteristics of traffic graph.

1) Sparsity. Traffic graphs are generally sparse. In a crossroad, one street connects other three streets and in an overpass, a street may connect more, but it only influences the constant we use here. Therefore, it is safe for us to assume that each node in the traffic graph has a degree of a small constant, for example 4, then the sum of all node degrees is $4*V$. Thus we get $E=(4*V)/2=2*V$. So for a traffic graph, the complexity of Bellman-Ford algorithm can be reduced to $O(2*V^2)$ which has the same order with Dijkstra's algorithm.
2) Hierarchy. In a traffic graph, a street only connects its neighboring streets. A hierarchy for the graph could be worked out by defining a division level based on the distance from the starting point. Due to the hierarchy characteristic, nodes that are closer to the starting node are more likely to influence other nodes. So, if updating operations on these nodes are implemented earlier, the time cost will reduce.

Considering the two characteristics of the traffic graphs, we propose several optimizations of Bellman-Ford algorithm for *Query_FiST* and *Query_BeST*, which can improve the efficiency significantly. The basic idea is to reduce the cost of updating

labels by replacing enumerating all edges with storing the updated nodes in heap or queue. Heap could provide a better updating sequence, while queue can handle the situation that distance labels on elements cannot be compared easily. The contributions of this paper include the following three aspects.

1) For the *Query_FiST*, we propose Heap-based Bellman-Ford algorithm to find the shortest path in a dynamically changing traffic graph and it works efficiently in practical implementations. Although in the worst case, the time complexity of our algorithm is worse than that of algorithms based on Dijkstra's algorithm, our algorithm works better in practical performance by taking advantages of the two characteristics of traffic graphs to avoid the worst case.
2) For the *Query_BeST*, we propose Extended Bellman-Ford algorithm to discover a shortest path with the best starting time in the traffic graph. The time complexity of our algorithm is $O(kE*a(T))$, where $a(T)$ is the time cost to do several operations between two functions and is inevitable in every algorithm focusing on this problem. The proposed algorithm for *Query_BeST* performs better than other previous ones because the mentioned characteristics in traffic graphs guarantee a smaller constant k. The experiment results confirm this assertion.
3) A series of experiments are implemented and the results show that our algorithms outperform existing solutions in terms of efficiency respectively.

The rest of this paper is organized as follows. Related works are introduced in section II and the problems are defined in section III. Then section IV proposes two algorithms for *Query_FiST* and *Query_BeST* and discusses the correctness and complexity respectively. In Section V, the implementations of the proposed algorithms are described in detail, and the analyses of the experimental results as well as the comparison to previous algorithms are provided. Finally, we conclude our paper and discuss some related issues in Section VI.

II. Related Work

Actually, shortest path problem has been a long hot topic in academia in the last decades. Dijkstra's algorithm, one of the most famous algorithms in this field, solves the single-source shortest path problem using greedy idea with complexity of $O(V^2)$ [14]. A general optimization for Dijkstra's original algorithm based on Fibonacci heap is proposed in [15], and the complexity is reduced to $O(E+V*log(V))$. Bellman-Ford algorithm is another classical algorithm for static shortest path based on dynamic programming, which could be directly implemented on graphs with negative weights on edges.

These algorithms work very well in the static graph. However, they do not suit to the prediction of the shortest path for private cars where the time-dependent dynamic traffic graphs are needed. In recent years, shortest path problems in time-dependent traffic graphs attracted more and more researchers, especially the two queries *Query_FiST* and *Query_BeST*.

A. Solutions for Query_FiST

In graphs with FIFO property, this problem is an extension to static shortest path problem [21]. Based on Dijkstra's algorithm, an efficient algorithm for this problem was proposed by extending the weight values of nodes to a piecewise linear function [11]. It is a labeling algorithm where each node is labeled by the earliest possible arrival time at that node with the given starting time from source node. In this algorithm, x_k is the permanent label of node k with an initial value *null* indicating the node is not settled yet, and y_k is its temporary label. At each iteration, the node i with the smallest y_i among all unsettled nodes is settled and the value of y_i is assigned to x_i. The time complexity of algorithm is $O(V^2)$, and it is not efficient for sparse traffic graphs.

A bidirectional A* algorithm has also been proposed for *Query_FiST* [19], where the result of the backward Dijkstra's algorithm is used as a preprocessing for the evaluation function $l(u,e,t)$, which means the shortest path from u to e departing at t. As the graph has FIFO property, we have $l(u,e,t)=d(e,u)<=d(u,e,t)$, and this equation makes the A* search available. By the way, if $l(u,e,t)=0$, then the algorithm becomes Dijkstra's algorithm. Its time complexity is uncertain since the efficiency of this pruning method is hard to estimate.

Chen, Lam, Sumalee, Li, and Tam [34] discussed a related problem which could be regarded as backward *Query_FiST*. They

then propose two efficient A* algorithms to solve both forward and backward *Query_FiST*, and prove their optimality.

B. Solutions for Query_BeST

Query_BeST requires the algorithm to determine the best time to start for the purpose of a shortest travelling time. Several solutions have been proposed towards this problem in the last twenty years. Algorithms on this query can be categorized into three groups: label correcting, label setting [18], and A* search [13]. Meanwhile, the influence of different weight functions for the algorithms on *Query_BeST* is discussed perfectly in [27]. Comparation about recent algorithms are introduced in [33].

1) Label correcting algorithm

Orda and Rom proposed a label correcting algorithm based on Bellman-Ford algorithm to solve this problem with the time complexity of $O(nm*a(T))$, where n is the number of nodes, m is the number of edges and $a(T)$ is the cost required for each function operation [11]. In this algorithm, two functions of time are used to label the arrival time for a given starting time, where one is to label the earliest arrival time at a node, and another is to label the earliest arrival time at node from its neighboring node. In consideration of its time complexity, this algorithm is unsuitable for traffic applications because it requires to process huge graphs and needs response in time.

2) Label setting algorithm

Dijkstra's algorithm is the most representative one among label setting algorithms. An algorithm is proposed based on it with the time complexity of $O((n*log(n)+m)*a(T))$ and the space complexity of $O((n+m)*a(T))$, where n is the number of nodes, m is the number of edges and $a(T)$ is the cost required for each function operation [12]. This algorithm includes two steps: time refinement and path selection. In the first step, the earliest arrival time for each node is calculated and the best starting time t^* is then calculated for the minimum travelling time. In the second step, one of the paths is selected from v_s to v_e, which matches the optimal travelling time. In their experiments, traffic graphs are generated using data from US Census Bureau 2005TIGER/Line. The nodes represent the starts, ends, and intersections of roads, while the edges represent road segments. And the edge value functions $w_{ij}(t)$ are generated randomly.

Toward graphs whose edge weights are piecewise linear functions, an algorithm is proposed to calculate the final earliest arrival time function $EAT_{sd}(t)$ directly and the time complexity is $O((F_d+\gamma)(E+V*log(V)))$, where F_d is the number of pieces in $EAT_{sd}(t)$ and γ is the total number of pieces in all edge weight functions in [20]. The efficiency of this algorithm depends on the output size of the final answer. In some cases where all edge weight functions have appropriate input sizes, it will be better than other algorithms. A new approach to model weighted graphs with correlated weights at the edges is proposed in [35], which is important to describe real world problems like routing in computer networks or finding shortest paths in traffic models under realistic assumptions.

3) A* search algorithm

An A* algorithm was proposed in [13] and its main idea is to maintain a priority queue of expanded paths, each of which starts from v_s. For each path $v_s \rightarrow v_k$, we maintain its cost function $f(k,t)=g(k,t)+h(k,t)$, which is a piecewise function of $t \in T$. $g(k,t)$ is the arrival time from source v_s to v_k along path p_k for starting at t; $h(k,t)$ is a lower bound estimation of the travelling time from v_k to destination v_e. In each iteration, the path whose cost function's minimum value during T is the smallest among all paths was picked out and then extended. The first path ending at v_e that is picked from the priority queue is the answer to the query. A* algorithm is more efficient only when the cost function could assist pruning the searching space efficiently, but its worst time complexity is imponderable.

III. PROBLEM DEFINATION

A. Construction of traffic graph

A traffic graph reflects the structure of the real road system of a city and the traffic time cost of each edge (through attaching a weight function of time). There are two different methods to construct the traffic graph.

1) Construction from the real topology of the real city road system. Points on the streets like landmarks, starting points, or terminal points, are chosen as nodes, and the segments between two nodes are regarded as edges. Once the nodes and edges are determined, the weight functions for each edge can be calculated using the traffic prediction algorithms according to the traffic data. Fig.1 shows the construction of the traffic graph. Fig.1(a) is a segment of the Shanghai road map, and Fig.1(b) is the traffic graph constructed by the above method, where *N1*, *N2*(the two ends of the Nanpu Bridge respectively), *N3*(a famous hotel) and *N4*(a big park) are selected as landmark nodes, and other nodes labeled in Fig.1(a) are selected because they are ends of some streets. The nodes in Fig.1(b) are connected according to the road map.

   Obviously, the topology structure constructed in this way is accurate. Although the transport bureau who has the road information can apply this method, it is hard for us to get such a perfect traffic topology in our experiments.

2) Construction from history traffic records. Firstly, the city road map is divided into meshes. Each cell can be viewed as a node initially. The connections between nodes can be extracted from the real records of the history traffic database. Once a car travelled from one cell to another, an edge between them should be added to the graph. Then, the duplicate edges between two nodes could be merged into one and self-loops on a node should be removed. In the final version of the traffic graph, the nodes should be removed if their degree is 0. Fig.2 shows the construction of this method with the same segment map of Fig.1(a). In Fig.2(a), the segment is divided into 25 cells and finally only 17 of them are selected as nodes in Fig. 2(b) and other 8 cells are removed for no car has visited these areas according to the history data.

   From the above graphs, we can see that if there are enough history traffic data and the lattices are drawn small enough, 100(m)*100(m) for example, the topology could be maintained very well. Therefore, this method is an easy and perfect alternative way for a traffic graph. Our experiment graph is based on this method.

B. Basic definition

A time dependent traffic graph can be described as $G(V, E, W)$, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, and W is the set of edge weight functions. For an edge $e_{ij}=(v_i,v_j)$, the weight function $w_{ij}(t)$ means the time cost travelling from v_i to v_j starting at time point t, where t is a time variable in the time domain T. A path between a starting node v_s and a terminal node v_e could be denoted as an edge sequence $p=e_1e_2\ldots e_{k-1}=(v_1,v_2)(v_2,v_3)\ldots(v_{k-1},v_k)$, where $v_1=v_s$, $v_k=v_e$. We use $fa(v)$ to represent node v's father node, which means there's an edge $(fa(v), v)$ in the path. For each node v_i in this sequence, there is an arrival time $arrive(v_i)$, a depart time $depart(v_i)$, and a waiting time $waiting(v_i)$. For a fixed starting time t, we have:

$arrive(v_1) = t$

$depart(v_i)=arrive(v_i)+waiting(v_i)$

Then we could define path's arrival time function $at(p, t) = arrive(v_e)$.

Given a time-dependent traffic graph $G_T(V, E, W)$, $Query\_FiST(v_s,v_e,st)$ means to find p^*, which satisfies:

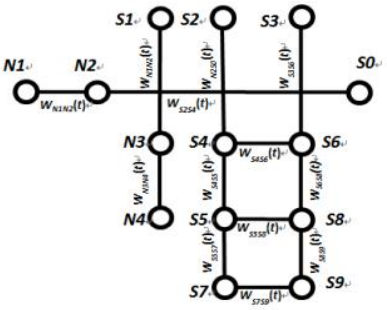
$at(p^*, st) = \min_p\{at(p, st)\}$

where st is the fixed starting time and $vs, ve \in V$ are starting node and terminal node.

While $Query\_BeST(v_s,v_e,st_1,st_2)$ means to find best starting time t*, which satisfies:

$at(p', t^*) - t^* = \min_{p,t}\{at(p, t) - t\}$, where p' is the shortest path if starting time is t* and st_1 is the earliest time you can choose to start and st_2 is the latest one.

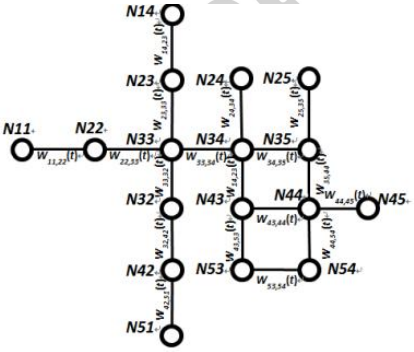
(a)



(b)

Fig. 1. Construction of the traffic graph from the real topology structure

(a)



(b)

Fig. 2. Construction of the traffic graph from the history traffic data

IV. ALGORITHM

The FIFO property suggests that the one who departs earlier from the starting point of an edge would reach the end of the edge earlier. We can state this property in a more general way. Time dependent graph $G_T(V, E, W)$ has FIFO property, if and only if every edge in the graph has FIFO property. An edge has FIFO property, if and only if for $t_\triangle >= 0$, $W_{uv}(t_0)+t_0 <= W_{uv}(t_0+t_\triangle)+(t_0+t_\triangle)$.

Actually, for a time dependent graph with FIFO property, there is the shortest path in which no waiting time exists. This property can be proved easily using reduction to absurdity. Assume u is a node where waiting is necessary to improve the arrival time, v is its successor, and the waiting time is wt. From the shortest path's property, we conclude that $W_{uv}(arrive(u)+w_t)+(arrive(u)+w_t) <= W_{uv}(arrive(u))+arrive(u)$, which contradicts to the graph's FIFO property. So the assumption is incorrect. In the following paper, $depart(v_i)=arrive(v_i)$ is adopted for the waiting time $waiting(v_i)$ could be neglected according to this property.

Moreover, "*relax*" is the primary operation in traditional Bellman-Ford algorithm, which corrects the distance label or earliest arrival time label $arrive(v_i)$ in our algorithm. The "*relax*" operation has two basic characteristics: 1) a "*relax*" operation could only be triggered by relaxed nodes; and 2) any node will not be relaxed too many times. The traditional Bellman-Ford algorithm enumerates all edges to "*relax*". We hope to improve the performance by only enumerating the edges of certain nodes, which has the potential to "*relax*" other nodes. These nodes are stored in heap or queue, which depends on the practical situation.

All shortest paths starting from v_s would be calculated when Bellman-Ford algorithm terminated, which indicates a lot of unnecessary calculations. As we only care about the shortest path from v_s to v_e, nodes located far from v_s and v_e should not be considered in the calculation. To avoid this, an upper bound ub for the shortest path should be added.

In our algorithms, *ub* indicates the time cost if drivers chose the shortest path based on space distance. To obtain the value of *ub*, a preprocessing should be applied in advance. As traffic graphs have no negative edges, Johnson's algorithm is not necessary. An obvious method to complete this preprocessing is to apply Dijkstra's algorithm $|V|$ times. The time complexity of this method is $O(V^2 log(V))$ and space complexity is $O(V^2)$. In the preprocessing, *prepath*[*s*][*j*] indicates the father node of v_j in the shortest path from v_s to v_j. Because the shortest path problem has the property of optimal substructure, every shortest path starting from v_i could be generated based on *prepath*[*i*]. The time cost of this shortest path in current traffic condition could be calculated and *ub* is generated.

This preprocessing should be applied only when there's a change in the traffic graph's topology. It guarantees that the shortest path discovered in our algorithm must be better than the one based on space distance because all nodes that would lead to longer path will not be added into the heap or the queue.

A. Heap-Based Bellman-Ford Algorithm

This section aims to answer *Query_FiST*(v_s,v_e,*st*). Considering the hierarchy characteristics of traffic map, the priority queue is put into use to store the nodes which have the potential to update other nodes' earliest arrival time. As the priority queue is essentially a heap, the time complexity of our operations, such as *enqueue*(*Q*,*v*) or *dequeue*(*Q*), is O(*log*(*V*)) where *V* is the number of nodes in the priority queue.

Using priority queue, a kind of common situation could be gotten rid of in a hierarchical graph. As shown in Fig.3, the path *N1-N2-N5* has a longer time cost than path *N1-N3-N4-N5*. For a normal queue, a bigger value of *arrive*(5) using *arrive*(2) can be firstly calculated and then other nodes adjacent to node *N5* would be relaxed. But as the program continues, *arrive*(5) would be relaxed again with path *N1-N3-N4-N5*, which leads to node *N5*'s *enqueue* operation again. As a result, all nodes that are adjacent to *N5* would be relaxed again. As a node with the minimum value of *arrive*(v_i) has a higher probability to reach the final status, we use the priority queue to let these nodes pop out quickly.

1) Algorithm description

During the algorithm's processing, a priority queue *Q* is maintained, which holds all relaxed nodes that could be used to relax other nodes adjacent to it and the key value used to modify the heap is the value of *arrive*(v_i). We use min-heap to maintain that the node v_i that has the minimum arrival time would be pop out firstly. At the beginning, the starting node is inserted into the heap. Then the algorithm comes into a loop till *Q* is empty.

Heap-based Bellman-Ford Algorithm
(1) for all $v_i \in V$, set *arrive*(v_i)=∞, *fa*(v_i)=-1 ;
(2) *arrive*(v_s)=*st* , *fa*(v_s)=v_s, generate *ub;*
(3) *clear*(*Q*), *enqueue*(*Q*, v_s);
(4) repeat
(5) v_c=*dequeue*(*Q*);
(6) for all v_i, (v_c,v_i)∈*E*
(7) if *arrive*(v_i)>*arrive*(v_c)+*Wci*(*arrive*(v_c)) then
(8) *arrive*(v_i)=*arrive*(v_c)+*Wci*(*arrive*(v_c)),
(9) *fa*(v_i)=v_c;
(10) if $v_i \notin Q$ and *arrive*(v_i) <*ub* then
(11) *enqueue*(*Q*, v_i)
(12) until *Q* is empty
(13) return *arrive*(v_e)

Fig. 4. Heap based Bellman-Ford Algorithm

At each iteration, the top node is popped out from the heap and relaxes all its neighboring nodes. If a node is relaxed successfully and its value is less than *ub*, then it will be inserted into the heap. The whole algorithm is outlined in Fig.4. $W_{ij}(t)$

denotes the time cost travelling from node v_i to v_j if departs v_i at time t, while $fa(v_i)$ is used to store the parent node of v_i, which will be used to generate the shortest path.

When the above algorithm is finished, the shortest path could be generated with the help of $fa(v)$ using the following algorithm in Fig.5. The shortest path is stored in array *path*. The operation *reverse* is used to modify the sequence of *path* to start as v_s.

Path generating Algorithm

(1) cnt=0, $path[cnt]$=v_e,cnt=cnt+1,tmp=v_e;
(2) repeat
(3) tmp=$fa[tmp]$, $path[cnt]$=tmp, cnt=cnt+1;
(4) until $fa[tmp]$=tmp
(5) *reverse*(*path*)
(6) return;

Fig. 5. Path generating algorithm

2) Correctness

Theorem 1. The path found in a traffic graph using Heap-based Bellman-Ford algorithm is the shortest path.

Proof. The priority queue in the algorithm is used to hold all nodes that could update other nodes' earliest arrival time. When the priority queue is empty, the algorithm will terminate, which indicates that no node could be relaxed. Suppose the path we found is not the shortest one, which means there exists a node whose arrival time is not the optimal one and an adjacent node could relax it. This conclusion contradicts our algorithm's termination condition. So, when the algorithm terminates, all nodes find its optimal arrival time and the shortest path is found.

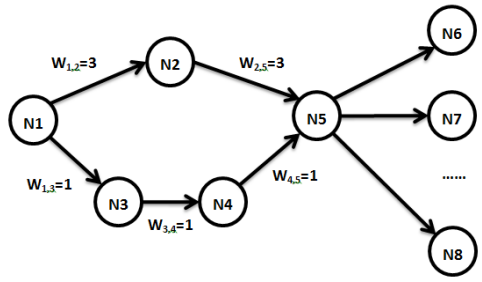


Fig. 3.Common situations on a hierarchy graph

Theorem 2. Heap-based Bellman-Ford algorithm will not form an endless loop and will terminate normally.

Proof. The loop termination condition in Heap-based Bellman-Ford algorithm is the priority queue becomes empty, which means the shortest path is found. At each iteration, an element must be popped out while it is not necessary to add elements to the queue. Only the element, which has been relaxed, should be added to the queue. Therefore, it is sure that there is a node whose earliest arrival time decreases at each iteration. Then we can conclude that, if the earliest path do exists and is finite, it can be discovered through finite iterations.

3) Complexity analysis

Let V be the nodes number of the graph, the average degree for each node is E/V. If the "while" loop executes m times, the total time complexity is $O(m*E/V)=O(m/V*E)$. For a heap, the complexity of the operations *dequeue* and *enqueue* is $O(log(V))$. Therefore, the time complexity of Heap-based Bellman-Ford algorithm is $O((m/V)*E*log(V))$. For the number m, it is really hard to be accurately calculated before the algorithm's execution. However, according to the distribution of the number of edges in the graph, m/V would be a constant for almost all cases. Let $k= m/V$, the experiments show that k is less than 2 (see the appendix).

Thus, for most cases, the time complexity of Heap-based Bellman-Ford algorithm is $O(kE*log(V))$ where E is the number of edges, V is the number of nodes and k is a constant.

B. Extended Bellman-Ford Algorithm

In this section, $Query\_BeST(v_s,v_e,st_1,st_2)$ would be solved. Given a starting time interval $T=[st_1,st_2]$ and the start node v_s and terminal node v_e. We will find the optimal t^* which makes $arrive(t^*)-t^*=\min_{t\in T}\{arrive(t)-t\}$. $ub(t)$ indicates the time cost if drivers start at time point t and following the shortest path based on distance.

1) Algorithm description

The framework of Extended Bellman-Ford algorithm is basically similar to that of Heap based Bellman-Ford algorithm. In this case, *arrive*(*v*) used in Heap-based Bellman-Ford algorithm needs to be developed. Instead of representing a single time, *arrive*(*v*) is required to store all the earliest times of different starting time. So, let function *arrive*(*v*,*t*) be the earliest arrival time at node *v*, from source v_s, for starting time *t*. Although the container to store the relaxed nodes becomes a normal queue because we could not compare two functions easily, our algorithm still avoids many unnecessary operations. The developed *relax* operation could be denoted as $arrive(v_i,t)=min(arrive(v_k,t)+ W_{ki}(arrive(v_k,t)), arrive(v_i,t))$, which indicates that for all $t\in T$, we extract the minimum value from $arrive(v_i,t)$ and $arrive(v_k,t)+ W_{ki}(arrive(v_k,t))$ and then assign this value to $arrive(v_i,t)$. $W_{ki}(arrive(v_k,t))$ also indicates a different operation, which is a compounding operation about functions. There are three different operations on the edge weight functions here.

- Linear combination of two functions: $c_1w_1(t)+c_2w_2(t)$.
- Minimum of two functions: $min\{w_1(t), w_2(t)\}$.
- Compounding two functions: $f(g(t))$.

Extended Bellman-Ford Algorithm

(1) for all $v_i\in V$ set $arrive(v_i,t)=\infty$ for $t\in T$;
(2) for $t\in T$ set $arrive(v_s, t)=t$, *enqueue*(*Q*, $arrive(v_s, t)$);
(3) $ub=max(ub(t))$ for $t\in T$;
(4) repeat
(5) $arrive(v_k, t)=dequeue(Q)$;
(6) for all $v_i\in\{v_j| (v_k, v_j)\in E\}$ do
(7) $arrive(v_i, t)=min(arrive(v_k, t)+W_{ki}(arrive(v_k, t)), arrive(v_i, t))$;
(8) if $arrive(v_i, t)$ changed and $arrive(v_i, t)-t<ub(t)$
(9) *enqueue*(*Q*, $arrive(v_i, t)$)
(10) until *Q* is empty

Fig. 6. Extended Bellman-Ford Algorithm

2) Correctness and Complexity analysis

The proof of Extended Bellman-Ford algorithm is essentially similar to that of Heap-based Bellman-Ford algorithm. The differences between these algorithms are the meaning of *arrive*() and the container to store updated nodes. These differences do not influence the correctness.

What's more, if the time is discrete, then process of this algorithm is equivalent to executing a Queue-based Bellman-Ford algorithm *T* times, which is obviously correct.

The Extended Bellman-Ford algorithm uses normal queue, so the time complexity of operations executed on heap is omitted. But the operations between two functions adds additional complexity, as is described before, we define it as $O(a(T))$. Therefore, the final time complexity of Extended Bellman-Ford algorithm is $O(kEa(T))$.

V. EXPERIMENT

A. Description

We have implemented the two proposed algorithms for *Query_Fist* and *Query_Best* on an open data set of real traffic. These data are provided by Wireless and Sensor networks Lab (WnSN), Shanghai Jiao Tong University [26]. In the dataset, the location information of about 4000 taxies is recorded at every 40 seconds within an area of 102 km^2 for 28 days(4 weeks). As it's hard to obtain the topology of Shanghai's road system, the whole Shanghai downtown area is divided into meshes to construct the traffic graph according to the latter method discussed in Section III. And connections between nodes are extracted from these real records. If there exists a taxi that has a path travelling through node *a* to node *b*, then we consider there's an edge between nodes *a* and *b*. The time distribution of a certain edge is extracted from the record, too. We use the average speed of taxis traveled through that edge in a certain time slot to represent the average time cost of passing through that edge.

In the implementation, the time dependent graph is stored in an edge array *ArrayEdge*. An index array *head* is used to find the first edge of every node in the graph. For instance, the location of $E_{1,2}$ in *ArrayEdge* is stored in *head*[1] as it is the first edge of node *N1*. Once the first edge is found, all other edges started from *N1* will be found because all these edges connected to *N1* form a linked list. The structure is illustrated in Fig.7.

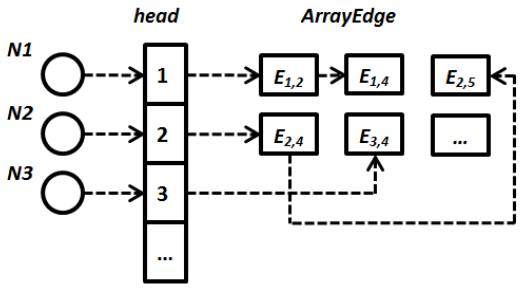


Fig. 7.Illustration of the edge array structure

All algorithms are realized using C++ and executed on a computer with 2.8 GHz Core i5 quad-core CPU, 4G memory PC, and Microsoft Windows 7.

B. Experiments on Query_FiST

To prove the contribution of a better path with the consideration of traffic condition, 10 random queries are raised and answered by our algorithm and the existing GPS algorithm, which considers the space information of roads only. TABLE I shows the travelling time of paths discovered by these two algorithms. It's obvious that our algorithm shortens the travelling time, averagely 40%.

A series of experiments have been carried out to compare our algorithms with other solutions. Traditional Orda's algorithm in [11] has a time complexity of $O(V^2)$. In our experiments, Orda's algorithm is optimized by using binary heap and its complexity has been reduced to $O(V*log(V))$ because the traditional one works too bad on the data set. Bellman-Ford algorithm is also selected to be implemented for the comparison.

The performance of all algorithms is measured from two aspects: the scale of node number and the scale of edge number. Six traffic graphs $G_{(1)}$~$G_{(6)}$ are generated by changing the mesh's size and described in TABLE II.

For each traffic graph listed in TABLE II, 20 random queries have been generated for testing and the results of the total time cost of the three algorithms for each traffic graph are shown in TABLE III.

Fig. 8 and Fig. 9 indicate the variation trend of time costs of the three algorithms according to the scale of nodes and edges respectively. We can see that our Heap-based Bellman-Ford algorithm enhances the performance of Bellman-Ford algorithm significantly. The time cost in last graph clearly shows that the increment speed of our algorithm is slower than that of Orda's algorithm. In G(6), this speed increment pattern is obvious. Compared to G(5), the time cost of Orda's algorithm increases more

than 7 times while the node number only increases 3 times in Fig. 8. Fortunately, the time cost of our algorithm is only 4 times longer.

C. *Experiments on Query_BeST*

We use the same dataset and set the starting time interval is [0, 50] in each query. Ding's algorithm [12] and A* search algorithm in [13] are implemented for comparation. As the time complexity of these algorithms is larger than that of algorithms solving *Query_FiST*, we generate another five traffic graphs with larger meshes $G^{(1)}$~$G^{(5)}$ as described in TABLE IV. TABLE V shows the average time costs of the three algorithms' execution. *RE* means memory overflow occurs and the execution of the program terminates abnormally.

Results of this experiment indicate that A* search algorithms works well on small graphs, when the node number is less than 1K and the edge number is less than 3K, but when the graph grows larger and larger, its time complexity grows exponentially. In the Fig.10 and Fig.11, the calculations cannot complete as its searching space also grows exponentially beyond the provide memory. Our algorithm shares the same increment speed with Ding's algorithm but the average time cost is less than that of ding's algorithm in every graph, approximately 3 times faster in each case.

To test the performance of algorithms according to different length of starting interval, 10 random queries in $G^{(3)}$ are generated. And the average time cost (second) is outlined in TABLE VI.

It is shown that the length of starting time interval influences the time cost, because longer interval means more time cost for function operations. From Fig. 12 we could see that our algorithm also outperforms other algorithms in this experiment.

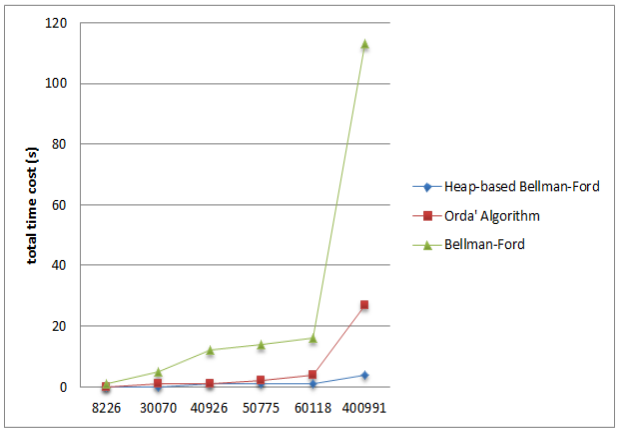


Fig. 8. Time costs of three algorithms on edge number

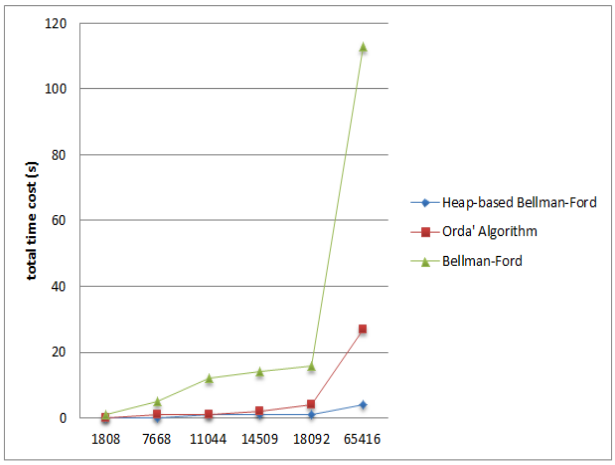


Fig. 9. Time costs of three algorithms on node number

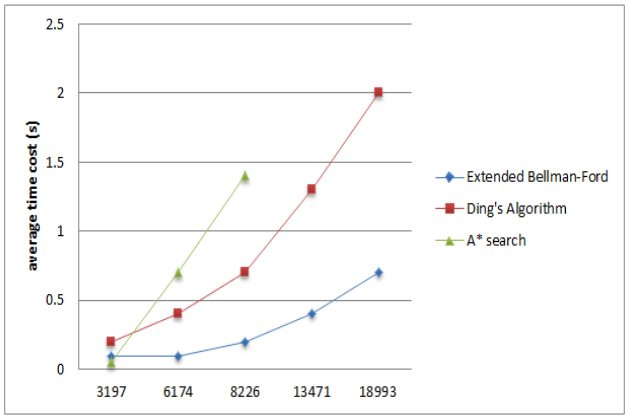


Fig. 10. Average time costs of three algorithms on edge number

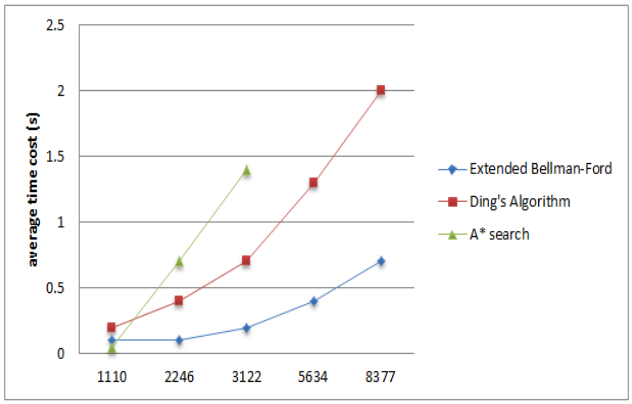


Fig. 11. Average time costs of three algorithms on node number

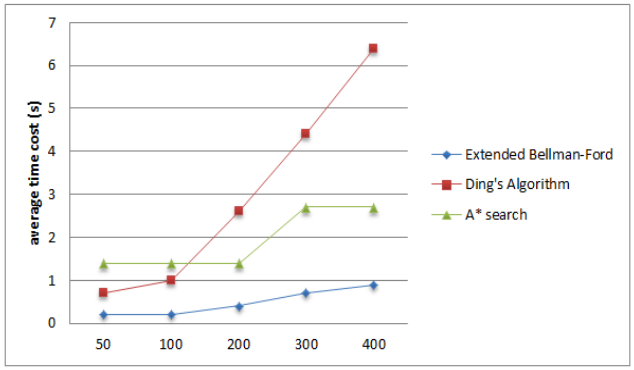


Fig. 12. Average time costs of three algorithms on interval length

VI. Conclusion

A traffic graph generated from history traffic information reflects the dynamic traffic status which can be used to predict the future traffic of a given stretch of the road and to discover the shortest path for drivers, thus supporting and facilitating green transportation. Two different types of shortest path queries are defined: *Query_FiST* for starting at a fixed time and hoping to reach the destination as early as possible, and *Query_BeST* for choosing a best starting time to avoid the rush hours.

Based on Bellman-Ford algorithm, we propose two algorithms: Heap-based Bellman-Ford algorithm for *Query_FiST* with the time complexity of $O(kE*log(V))$. Extended Bellman-Ford algorithm for *Query_BeST* with the time complexity of $O(kEa(T))$. To avoid unnecessary calculation, a preprocessing with time complexity of $O(V^2log(V))$ should be applied. A series experiments have been implemented on an open data set of real traffic collected from 4000+ taxis for 28 days. The experiments results show

that our algorithm outperform existing solutions obviously. The results also show that the processing time of all previous solutions is unacceptable when graphs have more than 10K nodes and 20K edges while both of our proposed algorithms can be applied into the practical systems for the average cost time is less than one second on an ordinary PC in a scale of 10K nodes and 20K edges.

Moreover, for larger traffic graphs, a lot of hierarchy model were proposed to guarantee efficient response. By preprocessing the shortest path between border nodes of different fragments, a query could be answered by combining this result with shortest paths within fragment. In [22], N. Jing, Y.-W. Huang, and E. A. Rundensteiner employed a path view materialization strategy. And in [23], S. Jung and S. Pramanik used a different graph partition method, which is disjointing node-set partition. These techniques also can be used into our algorithms when the traffic graph becomes too large to response immediately.

Besides the optimization focusing on algorithm, there're also many other attempts that also improve the speed of finding shortest path on a time dependent traffic graph. In [24], Jiang analyzed and did several experiments about the I/O efficiency of several shortest path algorithms in certain memory condition. In [25], a storage optimization towards spatial information is proposed.

ACKNOWLEDGMENTS

This research is sponsored by the National Natural Science Foundation of China (61371185, 61571049 , and 61171014).

REFERENCES

[1] Rajesh Krishna Balan, KhoaXuan Nguyen, Lingxiao Jiang, "Real-time trip information service for a large taxi fleet," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, June 28-July 01, 2011, Bethesda, Maryland, USA

[2] F. Li, Y. Yu, H. Lin, and W. Min. "Public bus arrival time prediction based on traffic information management system," in *Proceedings of IEEE international Conference on Service Operations and Logistics, and Informatics (SOLI)*, pages 336–341,2011.

[3] Pengfei Zhou, Yuanqing Zheng, Mo Li, "How long to wait?: predicting bus arrival time with mobile phone based participatory sensing,"in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, June 25-29, 2012, Low Wood Bay, Lake District, UK

[4] Chen, G., Yang, X., An, J., and Zhang, D. "Bus-Arrival-Time Prediction Models: Link-Based and Section-Based."*J. Transp. Eng.*, 138(1), 60–66. (2012)

[5] Prashanth Mohan, Venkata N. Padmanabhan, Ramachandran Ramjee, Nericell,"rich monitoring of road and traffic conditions using mobile smart phones," in *Proceedings of the 6th ACM conference on Embedded network sensor systems*, November 05-07, 2008, Raleigh, NC, USA

[6] Sasank Reddy, Min Mun, Jeff Burke, Deborah Estrin, Mark Hansen, Mani Srivastava, "Using mobile phones to determine transportation modes,"*ACM Transactions on Sensor Networks (TOSN)*, v.6 n.2, p.1-27, February 2010

[7] Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.,*Introduction to Algorithms*. McGraw-Hill Higher Education, New York (2001)

[8] Jing Yuan , Yu Zheng , Xing Xie , Guangzhong Sun, "Driving with knowledge from the physical world," in*Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, August 21-24, 2011, San Diego, California, USA

[9] Kai Xing, Xiuzhen Cheng, RongfangBie, Bowu Zhang, Liusheng Huang,"Traffic clustering and online traffic prediction in vehicle networks: A social influence perspective,"*INFOCOM*, 2012, p.495-503.

[10] Orda, A., Rom, R.,"Minimum weight paths in time-dependent networks,"*Networks* 21, 295–319 (1991)

[11] Orda, A., Rom, R.,"Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length," J. ACM 37(3), 607–625 (1990)

[12] Bolin Ding , Jeffrey Xu Yu , Lu Qin, "Finding time-dependent shortest paths over large graphs,"in*Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, March 25-29, 2008, Nantes, France

[13] Kanoulas, E., Du, Y., Xia, T., Zhang, D.,"Finding fastest paths on a road network with speed patterns,"in*ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, p. 10. IEEE Computer Society, Washington (2006)

[14] DIJKSTRA, E.W,"A note on two problems in connexion with graphs,"*Numer. Math.* 1, 4 (Sept. 1959), 269-27 i.

[15] Fredman, Michael Lawrence; Tarjan, Robert E,"Fibonacci heaps and their uses in improved network optimization algorithms". *25th Annual Symposium on Foundations of Computer Science (IEEE)*: 338–346.

[16] R. Bellman,"On a routing problem,"*Quarterly of Applied Mathematics*, 16(1):87-90, 1958.

[17] Jochen Eisner, Stefan Funke, and Sabine Storandt, "Optimal Route Planning for Electric Vehicles," in *Proc. 25th Assoc. Advancement Artificial Intell. Conf.*, San Francisco, CA, 2011.

[18] Dean, B.C.,"Shortest paths in FIFO time-dependent networks: theory and algorithms," Technical report, MIT Department of Computer Science (2004)

[19] GiacomoNannicini, Daniel Delling, Leo Liberti, DominikSchultes, "Bidirectional A* search for time-dependent fast paths," in*Proceedings of the 7th international conference on Experimental algorithms*, p.334-346, May 30-June 01, 2008, Provincetown, MA, USA

[20] Dehne, F., Omran, M.T, Sack, J.R., "Shortest Paths in Time-Dependent FIFO networks,"*Algorithmica* (2012) 62:416-435

[21] D. Dreyfus,"An Appraisal of Some Shortest Path Algorithms," Technical Report RM-5433, Rand Corporation, Santa Monica, CA, 1967.

[22] Ning Jing, Yun-Wu Huang, Elke A. Rundensteiner, "Hierarchical optimization of optimal path finding for transportation applications," in*Proceedings of the fifth international conference on Information and knowledge management*, p.261-268, November 12-16, 1996, Rockville, Maryland, United States

[23] S. Jung and S. Pramanik. "Hiti graph model of topographical roadmaps in navigation systems,"in *ICDE*, 1996.

[24] B. Jiang. DE, "I/O-Efficiency of Shortest Path Algorithms: An Analysis," in *ICDE*, pages 12–19, 1992.

[25] Y.Huang, N.Jing, and E.Rundensteiner,"Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations," In *VLDB*, pages 396–405, 1997.

[26] http://wirelesslab.sjtu.edu.cn

[27] Luca Foschini, John Hershberger ,SubhashSuri, "On the complexity of time-dependent shortest paths,"in *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, p.327-341, January 23-25, 2011, San Francisco, California.

[28] Paolo Narváez, Kai-YeungSiu, Hong-Yi Tzeng, "New dynamic algorithms for shortest path tree computation," *IEEE/ACM Transactions on Networking (TON)*, v.8 n.6, p.734-746, Dec. 2000

[29] Paolo Narváez, Kai-YeungSiu, Hong-Yi Tzeng, "New dynamic SPT algorithm based on a ball-and-string model,"*IEEE/ACM Transactions on Networking (TON)*, v.9 n.6, p.706-718, December 2001

[30] Y. Tian, K. C. K. Lee, and W. C. Lee,"Monitoring minimum cost paths on road networks,"in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 217--226, 2009.

[31] CamilDemetrescu, Stefano Emiliozzi, Giuseppe F. Italiano, "Experimental analysis of dynamic all pairs shortest path algorithms,"in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, January 11-14, 2004, New Orleans, Louisiana

[32] SudipMisra, B. John Oommen, "An Efficient Dynamic Algorithm for Maintaining All-Pairs Shortest Paths in Stochastic Networks,"*IEEE Transactions on Computers*, v.55 n.6, p.686-702, June 2006.

[33] Casey, Bradley, et al. "Critical review of time-dependent shortest path algorithms: A multimodal trip planner perspective." *Transport Reviews* 34.4 (2014): 522-539.

[34] Chen, Bi Yu, et al. "Reliable shortest path problems in stochastic time-dependent networks." *Journal of Intelligent Transportation Systems* 18.2 (2014): 177-189.

[35] Peter Buchholz, Iryna Felko, PH-graphs for analyzing shortest path problems with correlated traveling timesOriginal Research ArticleComputers & Operations Research, Volume 59, July 2015, Pages 51-65.

GPS	*our algorithm*
26.8	17.7
11.5	9.0
56.3	40.7
34.7	23.8
36.0	17.0
10.7	7.5
35.0	14.5
67.0	37.3
33.3	17.7
2.8	2.8

TABLE I

TRAVELLING TIME OF DIFFERENT ALGORITHMS

TABLE II

DESCRIPTIONS OF THE SIX TRAFFIC GRAPHS FOR *QUERY_FIST*

	$G_{(1)}$	$G_{(2)}$	$G_{(3)}$	$G_{(4)}$	$G_{(5)}$	$G_{(6)}$
Node number	1808	7668	11044	14509	18092	65416
Edge number	8226	30070	40926	50775	60118	400991
Mesh size(m)	1000	333	250	200	167	100

TABLE III

TOTAL TIME COST (SECONDS) ON *QUERY_FIST*

	$G_{(1)}$	$G_{(2)}$	$G_{(3)}$	$G_{(4)}$	$G_{(5)}$	$G_{(6)}$
Heap-based Bellman-Ford	0.1	0.1	1	1	1	4
Orda's algorithm	0.1	1	1	2	4	27
Bellman-Ford algorithm	1	5	12	14	16	113

TABLE IV

DESCRIPTIONS OF TRAFFIC GRAPHS FOR *QUERY_BEST*

	$G^{(1)}$	$G^{(2)}$	$G^{(3)}$	$G^{(4)}$	$G^{(5)}$
Node number	1110	2246	3122	5634	8377
Edge number	3197	6174	8226	13471	18993
Mesh size (m)	2000	1250	1000	667	500

TABLE V

Average time cost (seconds) of the three algorithms on*Query_BeST*

	$G^{(1)}$	$G^{(2)}$	$G^{(3)}$	$G^{(4)}$	$G^{(5)}$
Extended Bellman-Ford	0.1	0.1	0.2	0.4	0.7
Ding's algorithm	0.2	0.4	0.7	1.3	2
A* search	0.05	0.7	1.4	*RE*	*RE*

TABLE VI

Average time cost (seconds) according to different starting interval

Interval Length	*50*	*100*	*200*	*300*	*400*
Extended Bellman-Ford	0.2	0.2	0.4	0.7	0.9
Ding's algorithm	0.7	1	2.6	4.4	6.4
A* search	1.4	1.4	1.4	2.7	2.7

Highlights of the paper No. JNCA-D-15-00188

Green transportation technologies that reduce fuel consumption, idling, and vehicle miles traveled while reducing acute congestion could play a significant role in reducing greenhouse gas emissions, particularly in major cities, around ports and freight hubs, and on major roads and corridors. The key to support green transportation is to discover real-time shortest path for drivers.

In this paper, Considering the sparsity and the hierarchy characteristic of the traffic graphs, we propose several optimizations of Bellman-Ford algorithm for *Query_FiST* and *Query_BeST*, which can improve the efficiency significantly. The basic idea is to reduce the cost of updating labels by replacing enumerating all edges with storing the updated nodes in heap or queue. Heap could provide a better updating sequence, while queue can handle the situation that distance labels on elements cannot be compared easily. The highlights of this paper include the following three aspects.

1) For the *Query_FiST*, we propose Heap-based Bellman-Ford algorithm to find the shortest path in a dynamically changing traffic graph and it works efficiently in practical implementations. Although in the worst case, the time complexity of our algorithm is worse than that of algorithms based on Dijkstra's algorithm, our algorithm works better in practical performance by taking advantages of the two characteristics of traffic graphs to avoid the worst case.

2) For the *Query_BeST*, we propose Extended Bellman-Ford algorithm to discover a shortest path with the best starting time in the traffic graph. The time complexity of our algorithm is $O(kE*a(T))$, where $a(T)$ is the time cost to do several operations between two functions and is inevitable in every algorithm focusing on this problem. The proposed algorithm for *Query_BeST* performs better than other previous ones because the mentioned characteristics in traffic graphs guarantee a smaller constant k. The experiment results confirm this assertion.

3) A series of experiments are implemented and the results show that our algorithms outperform existing solutions in terms of efficiency respectively.