Online Computation of Fastest Path in Time-Dependent Spatial Networks*

Ugur Demiryurek[1], Farnoush Banaei-Kashani[1], Cyrus Shahabi[1], and Anand Ranganathan[2]

[1] University of Southern California- Department of Computer Science
Los Angeles, CA USA
{demiryur,banaeika,shahabi}@usc.edu
[2] IBM T.J. Watson Research Center
Hawthorne, NY USA
aranganaus@ibm.com

Abstract. The problem of point-to-point fastest path computation in static spatial networks is extensively studied with many precomputation techniques proposed to speed-up the computation. Most of the existing approaches make the simplifying assumption that travel-times of the network edges are constant. However, with real-world spatial networks the edge travel-times are time-dependent, where the arrival-time to an edge determines the actual travel-time on the edge. In this paper, we study the online computation of fastest path in time-dependent spatial networks and present a technique which speeds-up the path computation. We show that our fastest path computation based on a bidirectional time-dependent A* search significantly improves the computation time and storage complexity. With extensive experiments using real data-sets (including a variety of large spatial networks with real traffic data) we demonstrate the efficacy of our proposed techniques for online fastest path computation.

1 Introduction

With the ever-growing popularity of online map applications and their wide deployment in mobile devices and car-navigation systems, an increasing number of users search for point-to-point fastest paths and the corresponding travel-times. On static road networks where edge costs are constant, this problem has been extensively studied and many efficient speed-up techniques have been developed to compute the fastest path in a matter of milliseconds (e.g., [27,31,28,29]). The static fastest path approaches make the simplifying assumption that the travel-time for each edge of the road network is constant (e.g., proportional to the length of the edge). However, in real-world the actual travel-time on a road segment heavily depends on the traffic congestion and, therefore, is a function of time i.e., *time-dependent*. For example, Figure 1 shows the variation

* This research has been funded in part by NSF grants IIS-0238560 (PECASE), IIS-0534761,IIS-0742811 and CNS-0831505 (CyberTrust), and in part from CENS and METRANS Transportation Center, under grants from USDOT and Caltrans.Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

D. Pfoser et al. (Eds.): SSTD 2011, LNCS 6849, pp. 92–111, 2011.

of travel-time (computed by averaging two-years of historical traffic sensor data) for a particular road segment of I-10 freeway in Los Angeles as a function of arrival-time to the segment. As shown, the travel-time changes with time (i.e, the time that one arrives at the segment entry determines the travel-time), and the change in travel-time is significant. For instance, from 8AM to 9AM the travel-time of the segment changes from 32 minutes to 18 minutes (a 45% decrease). By induction, one can observe that the time-dependent edge travel-times yield a considerable change in the actual fastest path between any pair of nodes throughout the day. Specifically, the fastest between a source and a destination node varies depending on the departure-time from the source. Unfortunately, all those techniques that assume constant edge weights fail to address the fastest path computation in real-world time-dependent spatial networks.

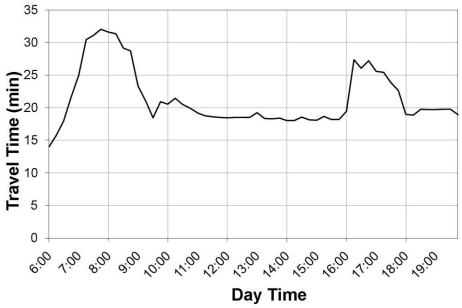


Fig. 1. Real-world travel-time on a segment of I-10 in LA

The time-dependent fastest path problem was first shown by Dreyfus [10] to be polynomially solvable in FIFO networks by a trivial modification to Dijkstra algorithm where, analogous to shortest path distances, the arrival-time to the nodes is used as the labels that form the basis of the greedy algorithm. The FIFO property, which typically holds for many networks including road networks, suggests that moving objects exit from an edge in the same order they entered the edge[1]. However, the modified Dijkstra algorithm [10] is far too slow for online map applications which are usually deployed on very large networks and require almost instant response times. On the other hand, there are many efficient precomputation approaches that answer fastest path queries in near real-time (e.g., [27]) in static road networks. However, it is infeasible to extend these approaches to time-dependent networks. This is because the input size (i.e., the number of fastest paths) increases drastically in time-dependent networks. Specifically, since the length of a s-d path changes depending on the departure-time from s, the fastest path is not unique for any pair of nodes in time-dependent networks. It has been conjectured in [3] and settled in [11] that the number of fastest paths between any pair of nodes in time-dependent road networks can be super-polynomial. Hence, an algorithm which considers the every possible path (corresponding to every possible departure-time from

[1] The fastest path computation is shown to be NP-hard in non-FIFO networks where waiting at nodes is not allowed [23]. Violation of the FIFO property rarely happens in real-world and hence is not the focus of this study.

the source) for any pair of nodes in large time-dependent networks would suffer from exponential time and prohibitively large storage requirements. For example, the time-dependent extension of Contraction Hierarchies (CH) [1] and SHARC [5] speed-up techniques (which are proved to be very efficient for static networks) suffer from the impractical precomputation times and intolerable storage complexity (see Section 3).

In this study, we propose a bidirectional time-dependent fastest path algorithm (B-TDFP) based on A* search [17]. There are two main challenges to employ bidirectional A* search in time-dependent networks. First, finding an admissible heuristic function (i.e., lower-bound distance) between an intermediate v_i node and the destination d is challenging as the distance between v_i and d changes based on the departure-time from v_i. Second, it is not possible to implement a backward search without knowing the arrival-time at the destination. We address the former challenge by partitioning the road network to non-overlapping partitions (an off-line operation) and precompute the intra (node-to-border) and inter (border-to-border) partition distance labels with respect to *Lower-bound Graph* $\underline{G}$ which is generated by substituting the edge travel-times in G with minimum possible travel-times. We use the combination of intra and inter distance labels as a heuristic function in the online computation. To address the latter challenge, we run the backward search on the lower-bound graph ($\underline{G}$) which enables us to filter-in the set of the nodes that needs to be explored by the forward search.

The remainder of this paper is organized as follows. In Section 2, we explain the importance of time-dependency for accurate and useful path planning. In Section 3, we review the related work on time-dependent fastest path algorithms. In Section 4, we formally define the time-dependent fastest path problem in spatial networks. In Section 5, we establish the theoretical foundation of our proposed bidirectional algorithm and explain our approach. In Section 6, we present the results of our experiments for both approaches with a variety of spatial networks with real-world time-dependent edge weights. Finally, in Section 7, we conclude and discuss our future work.

2 Towards Time-Dependent Path Planning

In this section, we explain the difference between fastest computation in time-dependent and static spatial networks. We also discuss the importance and the feasibility of time-dependent route planning.

To illustrate why classic fastest path computations in static road networks may return non-optimal results, we show a simple example in Figure 2 where a spatial network is modeled as a time-dependent graph and edge travel-times are function of time. Consider the snapshot of the network (i.e., a static network) with edge weights corresponding to travel-time values at t=0. With classic fastest path computation approaches that disregard time-dependent edge travel-times, the fastest path from s to d goes through v_1, v_2, v_4 with a cost of 13 time units. However, by the time when v_2 is reached (i.e., at t=5), the cost of edge $e(v_2, v_4)$ changes from 8 to 12 time units, and hence reaching d through v_2 takes 17 time units instead of 13 as it was anticipated at t=0. In contrast, if the time-dependency of edge travel-times are considered and hence the path going through v_3 was taken, the total travel-cost would have been 15 units which is the actual optimal fastest path. We call this shortcoming of the classic fastest path computation techniques as *no-lookahead* problem. Unfortunately, most of the existing state

of the art path planning applications (e.g., Google Maps, Bing Maps) suffer from the no-lookahead shortcoming and, hence, their fastest path recommendation remains the same throughout the day regardless of the departure-time from the source (i.e., query time). Although some of these applications provide alternative paths under traffic conditions (which may seem similar to time-dependent planning at first), we observe that the recommended alternative paths and their corresponding travel-times still remain unique during the day, and hence no time-dependent planning. To the best of our knowledge, these applications compute top-k fastest paths (i.e., k alternative paths) and their corresponding travel-times with and without taking into account the traffic conditions. The travel-times which take into account the traffic conditions are simply computed by considering increased edge weights (that corresponds to traffic congestion) for each path. However, our time-dependent path planning results in different optimum paths for different departure-times from the source. For example, consider Figure 3(a) where Google Maps offer two alternative paths (and their travel-times under no-traffic and traffic conditions) for an origin and destination pair in Los Angeles road network. Note that the path recommendation and the travel-times remain the same regardless of when the user submits the query. On the other hand, Figure 3(b) depicts the time-dependent path recommendations (in different colors for different departure times) for the same origin and destination pair where we computed the time-dependent fastest paths for 38 consecutive departure-times between 8AM and 5:30PM, spaced 15 minutes apart[2]. As shown, the optimal paths change frequently during the course of the day.

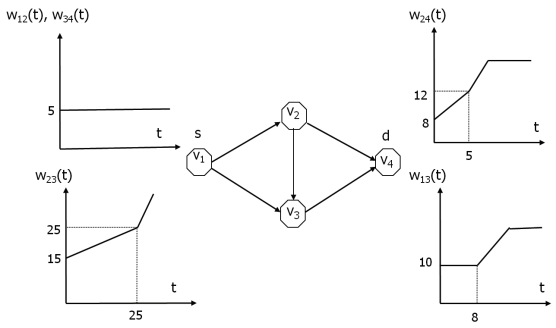
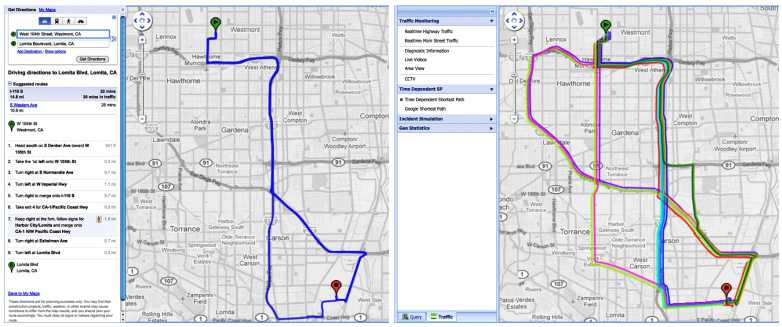


Fig. 2. Time-dependent graph

One may argue against the feasibility of time-dependent path planning algorithms due to a) unavailability of the time-dependent edge travel-times, or b) negligible gain of time-dependent path planning (i.e., how much time-dependent planning can improve the travel-time) over static path planning. To address the first argument, note that recent advances in sensor networks enabled instrumentation of road networks in major cities for collecting real-time traffic data, and hence it is now feasible to accurately model

[2] The paths are computed using the algorithm presented in Section 5 where time-dependent edge travel-times are generated based on the two-years of historical traffic sensor data collected from Los Angeles road network.

the time-dependent travel-times based on the vast amounts of historical data. For instance, at our research center, we maintain a very large traffic sensor dataset of Los Angeles County that we have been collecting and archiving the data for past two years (see Section 6.1 for the details of this dataset). As another example, PeMS [24] project developed by UC Berkeley generates time-varying edge travel-times using historical traffic sensor data throughout California. Meanwhile, we also witness that the leading navigation service providers (such as Navteq [22] and TeleAtlas [30]) started releasing their time-dependent travel-time data for road networks at high temporal resolution. With regards to the second argument, several recent studies showed the importance of time-dependent path planning in road networks where real-world traffic datasets have been used for the assessment. For example, in [7] we report that the fastest path computation that considers time-dependent edge travel-times in Los Angeles road network decreases the travel-time by as much as 68% over the fastest path computation that assumes constant edge travel-times. We made the similar observation in another study [15] under IBM's Smart Traffic Project where the time-dependent fastest path computation in Stockholm road network can improve the travel-time accuracy up to 62%. Considering the availability of high-resolution time-dependent travel-time data for road networks, and the importance of time-dependency for accurate and useful path planning, the need for efficient algorithms to enable next-generation time-dependent path planning applications becomes apparent and immediate.



(a) Static path planning (b) Time-dependent path planning

Fig. 3. Static vs Time-dependent path planning

3 Related Work

In the last decade, numerous efficient fastest path algorithms with precomputation methods have been proposed (see [29,27] for an overview). However, there are limited number of studies that focus on efficient computation of time-dependent fastest path (TDFP) problem.

Cooke and Halsey [2] first studied TDFP computation where they solved the problem using Dynamic Programming in discrete time. Another discrete-time solution to TDFP problem is to use time-expanded networks [19]. In general, time-expanded

network (TEN) and discrete-time approaches assume that the edge weight functions are defined over a finite discrete window of time $t \in t_0, t_1, .., t_n$, where t_n is determined by the total duration of time interval under consideration. Therefore, the problem is reduced to the problem of computing minimum-weight paths over a static static network per time window. Hence, one can apply any static fastest path algorithms to compute TDFP. Although these algorithms are easy to design and implement, they have numerous shortcomings. First, TEN models create a separate instance of network for each time instance hence yielding a substantial amount of storage overhead. Second, such approaches can only provide approximate results because the model misses the state of the network between any two discrete-time instants. Moreover, the difference between the shortest path obtained using TEN approach and the optimal shortest path is *unbounded*. This is because the query time can be always between any two of the intervals which are not captured by the model, and hence the error is is accumulated on each edge along the path. In [12], George and Shekhar proposed a time-aggregated graph approach where they aggregate the travel-times of each edge over the time instants into a time series. Their model requires less space than that of the TEN and the results are still approximate with no bounds.

In [10], Dreyfus showed that TDFP problem can be solved by a generalization of Dijkstra's method as efficiently as for static fastest path problems. However, Halpern [16] proved that the generalization of Dijkstra's algorithm is only true for FIFO networks. If the FIFO property does not hold in a time-dependent network, then the problem is NP-Hard. In [23], Orda and Rom introduced Bellman-Ford based algorithm where they determine the path toward destination by refining the arrival-time functions on each node in the whole time interval T. In [18], Kanoulas et al. proposed Time-Interval All Fastest Path (allFP) approach in which they maintain a priority queue of all paths to be expanded instead of sorting the priority queue by scalar values. Therefore, they enumerate all the paths from the source to a destination node which incurs exponential running time in the worst case. In [9], Ding et al. used a variation of Dijkstra's algorithm to solve the TDFP problem. With their TDFP algorithm, using Dijkstra like expansion, they decouple the path-selection and time-refinement (computing earliest arrival-time functions for nodes) for a given starting time interval T. Their algorithm is also shown to run in exponential time for special cases (see [4]). The focus of both [18] and [9] is to find the fastest path in time-dependent road networks for a given start time-interval (e.g., between 7:30AM and 8:30AM).

The ALT algorithm [13] was originally proposed to accelerate fastest path computation in static road networks. With ALT, a set of nodes called landmarks are chosen and then the shortest distances between all the nodes in the network and all the landmarks are computed and stored. ALT employs triangle inequality based on distances to the landmarks to obtain a heuristic function to be used in A* search. The time-dependent variant of this technique is studied in [6] (unidirectional) and [21] (bidirectional A* search) where heuristic function is computed w.r.t lower-bound graph. However, the landmark selection is very difficult (relies on heuristics) and the size of the search space is severely affected by the choice of landmarks. So far no optimal strategy with respect to landmark selection and random queries has been found. Specifically, landmark selection is NP-hard [26] and ALT does not guarantee to yield the smallest search spaces

with respect to fastest path computations where source and destination nodes are chosen at random. Our experiments with real-world time-dependent travel-times show that our approach consumes much less storage as compared to ALT based approaches and yields faster response times (see Section 6). In two different studies, The Contraction Hierarchies (CH) and SHARC methods (also developed for static networks) were augmented to time-dependent road networks in [1] and [5], respectively. The main idea of these techniques is to remove unimportant nodes from the graph without changing the fastest path distances between the remaining (more important) nodes. However, unlike the static networks, the importance of a node can change throughout the time under consideration in time-dependent networks, hence the importance of the nodes are time varying. Considering the super-polynomial input size (as discussed in Section 1), and hence the super-polynomial number of important nodes with time-dependent networks, the main shortcomings of these approaches are impractical preprocessing times and extensive space consumption. For example, the precomputation time for SHARC in time-dependent road networks takes more than 11 hours for relatively small road networks (e.g. LA with 304,162 nodes) [5]. Moreover, due to the significant use of arc flags [5], SHARC does not work in a dynamic scenario: whenever an edge cost function changes, arc flags should be recomputed, even though the graph partition need not be updated. While CH also suffers from slow preprocessing times, the space consumption for CH is at least 1000 bytes per node for less varied edge-weights where the storage cost increases with real-world time-dependent edge weights. Therefore, it may not be feasible to apply SHARC and CH to continental size road networks which can consist of more than 45 million road segments (e.g., North America road network) with possibly large varied edge-weights.

4 Problem Definition

There are various criteria to define the cost of a path in road networks. In our study we define the cost of a path as its travel-time. We model the road network as a *time-dependent weighted graph* as shown in Figure 2 where time-dependent travel-times are provided as a function of time which captures the typical congestion pattern for each segment of the road network. We use piecewise linear functions to represent the time-dependent travel-times in the network.

Definition 1. Time-dependent Graph. *A Time-dependent Graph is defined as* $G(V, E, T)$ *where* $V = \{v_i\}$ *is a set of nodes and* $E \subseteq V \times V$ *is a set of edges representing the network segments each connecting two nodes. For every edge* $e(v_i, v_j) \in E$*, and* $v_i \neq v_j$*, there is a cost function* $c_{v_i,v_j}(t)$*, where* t *is the time variable in time domain* T*. An edge cost function* $c_{v_i,v_j}(t)$ *specifies the travel-time from* v_i *to* v_j *starting at time* t*.*

Definition 2. Time-dependent Travel Cost. *Let* $\{s = v_1, v_2, ..., v_k = d\}$ *denotes a path which contains a sequence of nodes where* $e(v_i, v_{i+1}) \in E$ *and* $i = 1, ..., k-1$*. Given a* $G(V, E, T)$*, a path* $(s \rightsquigarrow d)$ *from source* s *to destination* d*, and a departure-time at the source* t_s*, the time-dependent travel cost* $TT(s \rightsquigarrow d, t_s)$ *is the time it takes to travel the path. Since the travel-time of an edge varies depending on the arrival-time to that edge, the travel-time of a path is computed as follows:*

$$TT(s \rightsquigarrow d, t_s) = \sum_{i=1}^{k-1} c_{v_i,v_{i+1}}(t_i) \text{ where } t_1 = t_s,\ t_{i+1} = t_i + c_{(v_i,v_{i+1})}(t_i),\ i = 1, .., k.$$

Definition 3. Lower-bound Graph. *Given a $G(V, E, T)$, the corresponding Lower-bound Graph $\underline{G}(V, E)$ is a graph with the same topology (i.e, nodes and edges) as graph G, where the weight of each edge c_{v_i,v_j} is fixed (not time-dependent) and is equal to the minimum possible weight $c^{min}_{v_i,v_j}$ where $\forall\ e(v_i, v_j) \in E, t \in T\ c^{min}_{v_i,v_j} \leq c_{v_i,v_j}(t)$.*

Definition 4. Lower-bound Travel Cost. *The lower-bound travel-time $LTT(s \rightsquigarrow d)$ of a path is less than the actual travel-time along that path and computed w.r.t $\underline{G}(V, E)$ as*

$$LTT(s \rightsquigarrow d) = \sum_{i=1}^{k-1} c^{min}_{v_i,v_{i+1}},\ i = 1, .., k.$$

It is important to note that for each source and destination pair (s, d), $LTT(s \rightsquigarrow d)$ is *time-independent* constant value and hence t is not included in its definition. Given the definitions of TT and LTT, the following property always holds for any path in $G(V, E, T)$: $LTT(s \rightsquigarrow d) \leq TT(s \rightsquigarrow d, t_s)$ where t_s is an arbitrary departure-time from s. We will use this property in subsequent sections to establish some properties of our proposed solution.

Definition 5. Time-dependent Fastest Path (TDFP). *Given a $G(V, E, T)$, s, d, and t_s, the time-dependent fastest path $TDFP(s, d, t_s)$ is a path with the minimum travel-time among all paths from s to d for starting time t_s.*

In the rest of this paper, we assume that $G(V, E, T)$ satisfies the First-In-First-Out (FIFO) property. We also assume that moving objects do not wait at any node. In most real-world applications, waiting at a node is not realistic as it means that the moving object must interrupt its travel by getting out of a road (e.g., exit freeway), and finding a place to park and wait.

5 Time-Dependent Fastest Path Computation

In this section, we explain our bidirectional time-dependent fastest path approach that we generalize bidirectional A* algorithm proposed for static spatial networks [25] to time-dependent road networks. Our proposed solution involves two phases. At the pre-computation phase, we partition the road network into non-overlapping partitions and precompute lower-bound distance labels within and across the partitions with respect to $\underline{G}(V, E)$. Successively, at the online phase, we use the precomputed distance labels as a heuristic function in our bidirectional time-dependent A* search that performs simultaneous searches from source and destination. Below we elaborate on both phases.

5.1 Precomputation Phase

The precomputation phase of our proposed algorithm includes two main steps in which we partition the road network into non-overlapping partitions and precompute lower-bound border-to-border, node-to-border, and border-to-node distance labels.

5.1.1 Road Network Partitioning

Real-world road networks are built on a well-defined hierarchy. For example, in United States, highways connect large regions such as states, interstate roads connect cities within a state, and multi-lane roads connect locations within a city. Almost all of the road network data providers (e.g., Navteq [22]) include road hierarchy information in their datasets. In this paper, we partition the graph to non-overlapping partitions by exploiting the predefined edge class information in road networks. Specifically, we first use higher level roads (e.g., interstate) to divide the road network into large regions. Then, we subdivide each large region using the next level roads and so on. We adopt this technique from [14] and note that our proposed algorithm is independent of the partitioning method, i.e., it yields correct results with all non-overlapping partitioning methods.

With our approach, we assume that the class of each edge $class(e)$ is predefined and we denote the class of a node $class(v)$ by the lowest class number of any incoming or outgoing edge to/from v. For instance, a node at the intersection of two freeway segments and an arterial road (i.e., the entry node to the freeway) is labeled with class of the freeway rather than the class of the arterial road. The input to our hierarchical partitioning method is the road network and the level of partitioning l. For example, if we like to partition a particular road network based on the interstates, freeways, and arterial roads in sequence, we set $l = 2$ where interstate edges represent the class 0. The road network partitions can be conceptually visualized as the areas after removal the nodes with $class(v) \leq l$ from $G(E, V)$.

Definition 6. *Given a graph $G(V, E)$, the partition of $G(V, E)$ is a set of subgraphs $\{S_1, S_2, ..., S_k\}$ where $S_i = (V_i, E_i)$ includes node set V_i where $V_i \cap V_j = \emptyset$ and $\cup_{i=1}^{k} V_i = V$, $i \neq j$.*

Given a $G(E, V)$ and level of partitioning l, we first assign to each node an empty set of partitions. Then, we choose a node v_i that is connected to edges other than the ones used for partitioning (i.e., a node with $class(v_i) > l$) and add partition number (e.g., S_1) to v_i's partition set. For instance, continuing with our example above, a node v_i with $class(v_i) > 2$ represent a particular node that belongs a less important road segment than an arterial road. Subsequently, we expand a shortest path tree from v_i to all it's neighbor nodes reachable through the edges of the classes greater than l, and add S_1 to their partition sets. Intuitively, we expand from v_i until we reach the roads that are used for partitioning. At this point we determine all the nodes that belong to S_1. Then, we select another node v_j with an empty partition set by adding the next partition number (e.g., S_2) to v_j's partition set and repeat the process. We terminate the process when all nodes are assigned to at least one partition. With this method we can easily find the border nodes for each partition, i.e., those nodes which include multiple partitions in their partition sets. Specifically, a node v, with $class(v) \leq l$ belongs to all partitions such that there is an edge e (with $class(e) > l$) connecting v to v' where $v' \in S_i$ and $i = 1, ..., k$, is the border node of the partitions that it connects to. Note that l is a tuning parameter in our partitioning method. Hence, one can arrange the size of the partitions by increasing or decreasing l.

Figure 4 shows the partitioning of San Joaquin (California) network based on the road classes. As shown, higher level edges are depicted with different (thicker) colors.

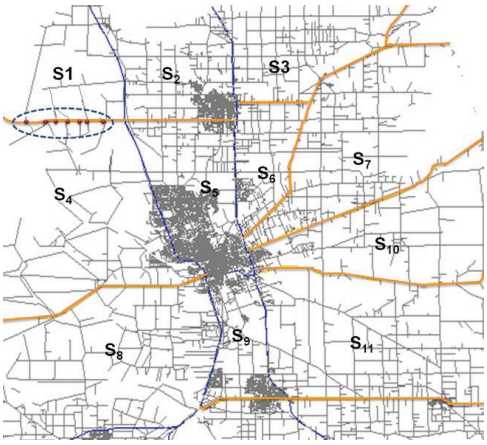


Fig. 4. Road network partitioning

Each partition is numbered starting from the north-west corner of the road network. The border nodes between partitions S_1 and S_4 are shown in the circled area. We remark that the number of border nodes (which can be potentially large depending on the density of the network) in the actual partitions have a negligible influence on the storage complexity. We explain the effect of the border nodes on the storage cost in the next section.

5.1.2 Distance Label Computation

In this step, for each pair of partitions (S_i,S_j) we compute the lower-bound fastest path cost w.r.t $\underline{G}$ between each border in S_i to each border node in S_j. However, we only store the minimum of all border-to-border fastest path distances. As an example, consider Figure 5 where the lower-bound fastest path cost between b_1 and b_3 (shown with straight line) is the minimum among all border-to-border distances (i.e., b_1-b_4, b_2-b_4, b_2-b_3) between S_1 and S_2. In addition, for each node v_i in a partition S_i, we compute the lower-bound fastest path cost from v_i to all border nodes in S_i w.r.t. $\underline{G}$ and store the minimum among them. We repeat the same process from border nodes in S_i to v_i. For example, border nodes b_1 and b_4 in Figure 5 are the nearest border nodes to s and d, respectively. We will use the precomputed node-to-border, border-to-border, and border-to-node lower-bound travel-times (referred to as distance labels) to construct our heuristic function for online time-dependent A* search. We used a similar distance label precomputation technique to expedite shortest path computation between network Voronoi polygons in static road networks [20].

We maintain the distance labels by attaching three attributes to each node representing a) the partition S_i that contains the node, b) minimum of the lower-bound distances from the node to border nodes, and c) minimum of the lower-bound distances from border nodes to the node (this is necessary for directed graphs). We keep border-to-border distance information in a hash table. Since we only store one distance value for each partition pair, the storage cost of the border-to-border distance labels is negligible.

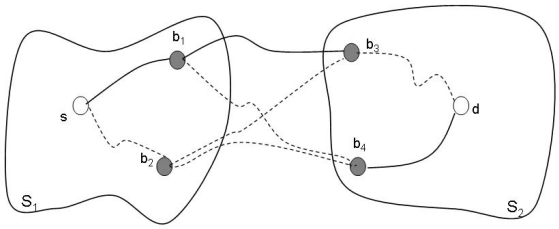


Fig. 5. Lower-bound distance computation

Another benefit of our proposed lower-bound computation is that the lower-bounds need to be updated when it is necessary. Specifically, we update the intra and inter distance labels only when the minimum travel-time of an edge changes, otherwise, the travel-time updates are discarded. Note that intra distance label computation is local, i.e., we only update the intra distance labels for the partitions in which the minimum travel-time of an edge changes.

5.2 Online B-TDFP Computation

As showed in [10], the time-dependent fastest path problem (in FIFO networks) can be solved by modifying Dijkstra algorithm. We refer to modified Dijkstra algorithm as time-dependent Dijkstra (TD-Dijkstra). TD-Dijkstra visits all network nodes reachable from s in every direction until destination node d is reached. On the other hand, a time-dependent A* algorithm can significantly reduce the number of nodes that have to be traversed in TD-Dijkstra algorithm by employing a heuristic function $h(v)$ that directs the search towards destination. To guarantee optimal results, $h(v)$ must be admissible and consistent (a.k.a, monotonic). The admissibility implies that $h(v)$ must be less than or equal to the actual distance between v and d. With static road networks where the length of an edge is constant, Euclidian distance between v and d is used as $h(v)$. However, this simple heuristic function cannot be directly applied to time-dependent road networks, because, the optimal travel-time between v and d changes based on the departure-time t_v from v. Therefore, in time-dependent road networks, we need to use an estimator that never overestimates the travel-time between v and d for any possible t_v. One simple lower-bound estimator is $d_{euc}(v, d)/max(speed)$, i.e., the Euclidean distance between v and d divided by the maximum speed among the edges in the entire network. Although this estimator is guaranteed to be a lower-bound, it is a very loose bound, and hence yields insignificant pruning.

With our approach, we obtain a much tighter bound by utilizing the precomputed distance labels. Assuming that an on-line time-dependent fastest path query requests a path from source s in partition S_i to destination d in partition S_j, the fastest path must pass through from one border node b_i in S_i and another border node b_j in S_j. We know that the time-dependent fastest path distance passing from b_i and b_j is greater than or equal to the precomputed lower-bound border-to-border (e.g., $LTT(b_l, b_t)$) distance for S_i and S_j pair. We also know that a time-dependent fastest path distance from s to b_i

is always greater than or equal to the precomputed lower-bound fastest path distance of s to its nearest border node b_s. Analogously, same is true from the border node b_d (i.e., nearest border node) to d in S_j. Thus, we can compute a lower-bound estimator of s by $h(s) = LTT(s, b_s) + LTT(b_l, b_t) + LTT(b_d, d)$.

Lemma 1. *Given an intermediate node v_i in S_i and destination node d in S_j, the estimator $h(v_i)$ is admissible, i.e., a lower-bound of time-dependent fastest path distance from v_i to d passing from border nodes b_i and b_j in S_i and S_j,respectively.*

Proof. Assume $LTT(b_l, b_t)$ is the minimum border-to-border distance between S_i and S_j, and b'_i, b'_j are the nearest border nodes to v_i and d in $\underline{G}$, respectively. By definition of $\underline{G}(V, E)$, $LTT(v_i, b'_i) \leq TDFP(v_i, b_i, t_{v_i})$, $LTT(b_l, b_t) \leq TDFP(b_i, b_j, t_{b_i})$, and $LTT(b'_j, d) \leq TDFP(b_j, d, t_{b_j})$ Then, we have $h(v_i) = LTT(v_i, b'_i) + LTT(b_l, b_t) + LTT(b'_j, d) \leq TDFP(v_i, b_i, t_{v_i}) + TDFP(b_i, b_j, t_{b_i}) + TDFP(b_j, d, t_{b_j})$

We can use our $h(v)$ heuristic with unidirectional time-dependent A* search in road networks. The time-dependent A* algorithm is a best-first search algorithm which scans nodes based on their time-dependent cost label (maintained in a priority queue) to source similar to [10]. The only difference to [10] is that the label within the priority queue is not determined only by the time-dependent distance to source but also by a lower-bound of the distance to d, i.e., $h(v)$ introduced above.

To further speed-up the computation, we propose a bidirectional search that simultaneously searches forward from the source and backwards from the destination until the search frontiers meet. However, bidirectional search is challenging in time-dependent road networks for two following reasons. First, it is essential to start the backward search from the arrival-time at the destination t_d and exact t_d cannot be evaluated in advance at the query time (recall that arrival-time to destination depends on the departure-time from the source in time-dependent road networks). We address this problem by running a backward A* search that is based on the reverse lower-bound graph $\overleftarrow{\underline{G}}$ (the lower-bound graph with every edge reversed). The main idea with running backward search in $\overleftarrow{\underline{G}}$ is to determine the set of nodes that will be explored by the forward A* search. Second, it is not straightforward to satisfy the consistency (the second optimality condition of A* search) of $h(v)$ as the forward and reverse searches use different distance functions. Next, we explain bidirectional time-dependent A* search algorithm (Algorithm 1) and how we satisfy the consistency.

Given $G = (V, E, T)$, s and d, and departure-time t_s from s, let Q_f and Q_b represent the two priority queues that maintain the labels of nodes to be processed with forward and backward A* search, respectively. Let F represent the set of nodes scanned by the forward search and N_f is the corresponding set of labeled vertices (those in its priority queue). We denote the label of a node in N_f by d_{fv}. Analogously, we define B, N_b, and d_{fv} for the backward search. Note that during the bidirectional search F and B are disjoint but N_f and N_b may intersect. We simultaneously run the forward and backward A* searches on $G(V, E, T)$ and $\overleftarrow{\underline{G}}$, respectively (Line 4 in Algorithm 1). We keep all the nodes visited by backward search in a set H (Line 5). When the search frontiers meet, i.e., as soon as N_f and N_b have a node u in common (Line 6), the cost of the time-dependent fastest path ($TDFP(s, u, t_s)$) from s to u is determined. At this

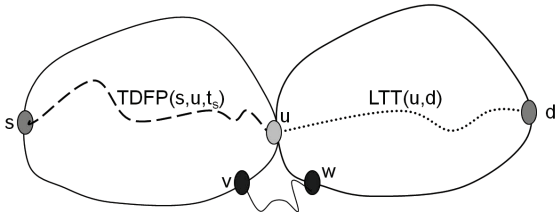


Fig. 6. Bidirectional search

point, we know that $TDFP(u, d, t_u) > LTT(u, d)$ for the path found by the backward search. Hence, the time-dependent cost of the paths (found so far) passing from u is the upper-bound of the time-dependent fastest path from s to d, i.e., $TDFP(s, u, t_s) + TDFP(u, d, t_u) \geq TDFP(s, d, t_s)$.

If we stop the searches as soon as a node u is scanned by both forward and backward searches, we cannot guarantee finding the time-dependent fastest path from u to d within the set of nodes in H. This is due to inconsistent potential function used in bidirectional search that relies on two independent potential functions for two inner A* algorithms. Specifically, let $h_f(v)$ (estimated distance from node v to target) and $h_b(v)$ (estimated distance from node v to source) be the potential functions used in the forward and backward searches, respectively. With the backward search, each original edge $e(i, j)$ considered as $e(j, i)$ in the reverse graph where h_b used as the potential function, and hence the reduced cost[3] of $e(j, i)$ w.r.t. h_b is computed by $c_{h_b}(j, i)$=$c(i, j)$-$h_b(j)$+$h_b(i)$ where $c(i, j)$ is the cost in the original graph. Note that h_f and h_b are consistent if, for all edges (i, j), $c_{h_f}(i, j)$ in the original graph is equal to $c_{h_b}(j, i)$ in the reverse graph. If h_f and h_b are not consistent, there is no guarantee that the shortest path can be found when the search frontiers meet. For instance, consider Figure 6 where the forward and backward searches meet at node u. As shown, if v is scanned before u by the forward search, then $TDFP(s, u, t_s) > TDFP(s, v, t_s)$. Similarly if w is scanned before u by the backward search, the $LTT(u, d) > LTT(w, d)$ and hence $TDFP(u, d, t_u) > TDFP(w, d, t_w)$. Consequently, it is possible that $TDFP(s, u, t_s) + TDFP(u, d, t_u) \geq TDFP(s, v, t_s) + TDFP(w, d, t_w)$. To address this challenge, one needs to find a) a consistent heuristic function and stop the search when the forward and backward searches meet or b) a new termination condition. In this study, we develop a new termination condition (the proof of correctness is given below) in which we continue both searches until the Q_b only contains nodes whose labels exceed $TDFP(s, u, t_s) + TDFP(u, d, t_u)$ by adding all visited nodes to H (Line 9-11). Recall that the label (denoted by d_{bv}) of node v in the backward search priority queue Q_b is computed by the time-dependent distance from the destination to v plus the lower-bound distance from v to s, i.e., $d_{bv} = TDFP(v, d, t_v) + h(v)$. Hence, we stop the search when $d_{bv} > TDFP(s, u, t_s) + TDFP(u, d, t_u)$. As we explained, $TDFP(s, u, t_s) + TDFP(u, d, t_u)$ is the length of the fastest path seen so far (not necessarily the actual fastest path) and is updated during the search when a new

[3] A* search is equivalent to Dijkstra's algorithm on a transformed network in which the cost of each edge $c(i, j)$ is equal to $c(i, j)$-$h(i)$+$h(j)$.

common node u' found with $TDFP(s,u',t_s)+TDFP(u',d,t_{u'}) < TDFP(s,u,t_s)+TDFP(u,d,t_u)$. Once both searches stop, H will include all the candidate nodes that can possibly be part of the time-dependent fastest path to d. Finally, we continue the forward search considering only the nodes in H until we reach d (Line 12).

Algorithm 1. B-TDFP Algorithm

1: //Input: G_T, $\overleftarrow{G}$, s:source, d:destination,t_s:departure time
2: //Output: a (s,d,t_s) fastest path
3: //$FS()$:forward search, $BS()$:backward search, N_f/N_b: nodes scanned by FS()/BS(), d_{bv}:label of the minimum element in BS queue
4: $FS(G_T)$ and $BS(\overleftarrow{G})$ //start searches simultaneously
5: $N_f \leftarrow FS(G_T)$ and $N_b \leftarrow BS(\overleftarrow{G})$
6: $If\ N_f \cap N_b \neq \emptyset\ then\ u \leftarrow N_f \cap N_b$
7: $M = TDFP(s,u,t_s) + TDFP(u,d,t_u)$
8: $end\ If$
9: $While\ d_{bv} \geq M$
10: $N_b \leftarrow BS(\overleftarrow{G})$
11: $End\ While$
12: $FS(N_b)$
13: $return\ (s,d,t_s)$

Lemma 2. *Algorithm 1 finds the correct time-dependent fastest path from source to destination for a given departure-time t_s.*

Proof. We prove Lemma 2 by contradiction. The forward search in Algorithm 1 is the same as the unidirectional A* algorithm and our heuristic function $h(v)$ is a lower-bound of time-dependent distance from u to v. Therefore, the forward search is correct. Now, let $P(s,(u),d,t_s)$ represent the path from s to d passing from u where forward and backward searches meet and ω denotes the cost of this path. As we showed ω is the upper-bound of actual time-dependent fastest path from s to d. Let ϕ be the smallest label of the backward search in priority queue Q_b when both forward and backward searches stopped. Recall that we stop searches when $\phi > \omega$. Suppose that Algorithm 1 is not correct and yields a suboptimal path, i.e., the fastest path passes from a node outside of the corridor generated by the forward and backward searches. Let $P*$ be the fastest path from s to d for departure-time t_s and cost of this path is α. Let v be the first node on $P*$ which is going to be explored by the forward search and not explored by the backward search and $h_b(v)$ is the heuristic function for the backward search. Hence, we have $\phi \leq h_b(v) + LTT(v,d)$, $\alpha \leq \omega < \phi$ and $h_b(v) + LTT(v,d) \leq LTT(s,v) + LTT(v,d) \leq TDFP(s,v,t_s)+TDFP(v,t,t_v) = \alpha$, which is a contradiction. Hence, the fastest path will be found in the corridor of the nodes labeled by the backward search.

6 Experimental Evaluation

6.1 Experimental Setup

We conducted extensive experiments with different spatial networks to evaluate the performance of our proposed bidirectional time-dependent fastest path (B-TDFP)

approach. As of our dataset, we used California (CA), Los Angeles (LA) and San Joaquin County (SJ) road network data (obtained from Navteq [22]) with approximately 1,965,300, 304,162 and 24,123 nodes, respectively. We conducted our experiments on a server with 2.7 GHz Pentium Core Duo processor with 12GB RAM memory.

6.1.1 Time-Dependent Network Modeling

At our research center, we maintain a very large-scale and high resolution (both spatial and temporal) traffic sensor (i.e., loop detector) dataset collected from entire LA County highways and arterial streets. This dataset includes both inventory and real-time data for 6300 traffic sensors covering approximately 3000 miles. The sampling rate of the streaming data is 1 reading/sensor/min. We have been continuously collecting and archiving the traffic sensor data for the past two years. We use this real-world dataset to create time varying edge weights; we spatially and temporally aggregate sensor data by assigning interpolation points (for each 5 minutes) that depict the travel-times on the network segments. Based on our observation, all roads are un-congested between 9PM and 6AM, and hence we assume static edge weights during this interval. In order to create time-dependent edge weights for the local streets in LA, CA and SJ, we developed a traffic modeling approach [8] that synthetically generates the edge travel-time profiles. Our approach uses spatial (e.g., locality, connectivity) and temporal (e.g., rush hour, weekday) characteristics to generate travel-time for network edges that does not have readily available sensor data.

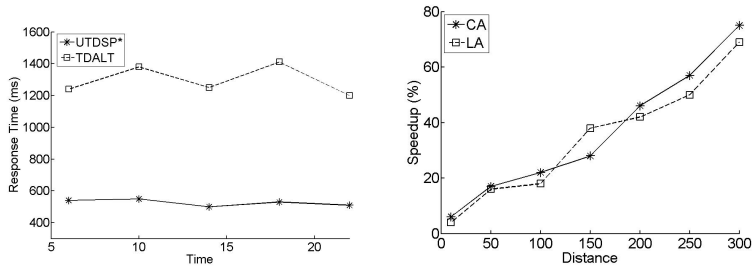
6.2 Results

In this section, we report the experimental results from our fastest path queries in which we determine the s and d nodes uniformly at random. We also pick our departure-time randomly and uniformly distributed in time domain T. The average results are derived from 1000 random s-d queries. We only present the results for LA and CA, the experimental results for both SJ and LA are very similar.

6.2.1 Comparison with ALT

In this set of experiments we compare our algorithm with time-dependent ALT (TD-ALT) approaches [6,21] with respect to storage and response time. We run our proposed algorithm both unidirectionally and bidirectionally (in CA network) and compare with [6] and [21], respectively. As we mentioned, selecting good landmarks that lead to good performance is very difficult and hence several heuristics have been proposed for landmark selection. Among these heuristics, we use the best known technique; maxCover (see [6]) with 64 landmarks. We computed travel-times between each node and the landmarks with respect to $\underline{\mathrm{G}}$. Under this setting, to store the precomputed distances, TD-ALT attaches to each node an array of 64 elements corresponding to the number of landmarks. Assuming that each array element takes 2 bytes of space, the additional storage requirement of TD-ALT is 63 Megabytes. On the other hand, with our algorithm, we divide CA network to 60 partitions and store the intra and inter distance labels. The total storage requirement of our proposed solution is 8.5 Megabytes where we consume, for each node, an array of 2 elements (corresponding to *from* and *to* distances

to the closest border node) plus the border-to-border distance labels. Since the experimental results for both unidirectional and bidirectional searches differ insignificantly and due to space limitations, we only present the results from unidirectional search below. As shown in Figure 7(a) the response time of our unidirectional time-dependent A* search (U-TDFP) is approximately three times better than that of TD-ALT for all times. This is because the search space of TD-ALT is severely affected by the quality of the landmarks which are selected based on a heuristic. Specifically, TD-ALT may yield very loose bounds based on the randomly selected s and d, and hence the large search space. In addition, with each iteration, TD-ALT needs to find the best landmark (among 64 landmarks) which yields largest triangular inequality distance for better pruning; it seems that the overhead of this operation is not negligible. On the other hand, U-TDFP yields a more directional search with the help of intra and inter distance labels with no additional computation.



(a) ALT vs U-TDFP (b) Speed-up ratio

Fig. 7. TD-ALT Comparison and Speed-up Ratio Analysis

6.2.2 Performance of B-TDFP

In this set of experiments, we compare the performance of our proposed approach to other existing TDFP methods w.r.t to a) preprocessing time, b) storage (byte per node), c) the average number of relaxed edges, and d) average query time. Table 1 shows the preprocessing time (Pre Processing), storage (Storage), number of scanned nodes (#Nodes), and response time (Res. Time) of time-dependent Dijkstra (TD-Dijkstra) implemented based on [10], unidirectional (U-TDFP) and bidirectional (B-TDFP) time-dependent A* search implemented using our proposed heuristic function, time-dependent Contraction Hierarchies (TD-CH) [1], and time-dependent SHARC (TD-SHARC) [5]. To implement U-TDFP and B-TDFP, we divide CA and LA network to 60 (which roughly correspond to counties in CA) and 25 partitions, respectively. Comparing TD-Dijkstra with our approach, we observe a very high trade-off between the query results and precomputation in both LA and CA networks. Our proposed B-TDFP performs 23 times better than TD-Dijkstra depending on the network while preprocessing and storage overhead is relatively small. As shown, the preprocessing time and storage complexity is directly proportional to network size.

Comparing the time-dependent variant of SHARC (TD-SHARC) and CH (TD-CH) with our approach, we observe B-TDFP outperforms TD-SHARC and TD-CH in preprocessing and response time. We also observe that as the graph gets bigger or more

Table 1. Experimental Results

	Algorithm	PreProcessing [h:m]	Storage [B/node]	#Nodes	Res. Time [ms]
CA	TD-Dijkstra	0:00	0	1162323	4104.11
	U-TDFP	1:13	6.82	90575	310.17
	B-TDFP	1:13	6.82	67172	182.06
	TD-SHARC	19:41	154.10	75104	227.26
	TD-CH	3:55	1018.33	70011	209.12
LA	TD-Dijkstra	0:00	0	210384	2590.07
	U-TDFP	0:27	3.51	11115	197.23
	B-TDFP	0:27	3.51	6681	101.22
	TD-SHARC	11:12	68.47	9566	168.11
	TD-CH	1:58	740.88	7922	140.25

edges are time-dependent, the preprocessing time of TD-SHARC increases drastically. The preprocessing of TD-SHARC takes very long for both road networks, i.e., up to 20 times more than B-TDFP. The reason for the performance gap is that TD-SHARC's contraction routine cannot bypass the majority of the nodes in time-dependent road networks as in the static road networks. Recall that the importance of a node can change throughout the time under consideration in time-dependent road networks. In addition, TD-SHARC is very sensitive to edge cost function changes, i.e. whenever cost function of an edge changes, the preprocessing phase needs to be repeated to determine the by-pass nodes. While TD-CH tend to have better response times than TD-SHARC, the space consumption of TD-CH is significantly high (approximately 1000 bytes per node in CA network). For this reason, TD-CH is not feasible for very large road networks such as North America and Europe. We note that, to improve the response and preprocessing time, several variations of TD-SHARC and TD-CH algorithms are implemented in the literature. These variations trade-off between the optimality of the solution and the response time. For example, the response time of Heuristic TD-SHARC [5] is shown much better than that of original TD-SHARC algorithm. However, the path found by the Heuristic TD-SHARC is not optimal and the error rate is not bounded. As another example, the performance of TD-SHARC can be improved by combining with another technique called Arc-Flags [5]. Similar performance improvements can be applied to our proposed approach. For instance, we can terminate the search when the search frontiers meet and report the combination of path found by the forward and backward search as the result. However, as mentioned in Section 5.2, we cannot guarantee the optimal solution in this setting. Moreover, based on our initial observation and implementation, we can also integrate our algorithm with Arc-Flags. However, the focus of our study is to develop a technique that yields exact solutions. Hence, for the sake of simplicity and

fair comparison, we only compare the original algorithms that yields exact results and do not consider integrating different methods.

6.2.3 Quality of Lower-Bounds

As discussed, the performance of time-dependent A* search depends on the lower-bound distance. In this set of experiments, we analyze the quality of our proposed lower-bound computed based on the Distance Labels explained in Section 5.1.2. We define the lower-bound quality by $lg = \frac{\delta(u,v)}{d(u,v)}$, where $\delta(u,v)$ and $d(u,v)$ represent the estimated and actual travel-times between nodes u and v, respectively. Table 2 reports lg based on three different heuristic function, namely Naive, ALT, and DL (i.e., our heuristic function computed based on Distance Labels). Similar to other experiments, the values in Table 2 are obtained by selecting s, d and t_s uniformly at random between 6AM and 9PM. We compute the naive lower-bound estimator by $\frac{d_{euc}(u,v)}{max(speed)}$, i.e., the Euclidean distance between u and v is divided by the maximum speed among the edges in the entire network. We obtain the ALT lower-bounds based on G and the maxCover ([6]) technique with 64 landmarks. As shown, DL provides better heuristic function in both LA and CA. The reason is that the ALT's lg relies on the distribution of the landmarks, and hence depending on the location of s and d it is possible to get very loose bounds. On the other hand, the lower-bounds computed based on Distance Labels are more directional. Specifically, with our approach the s and d nodes must reside in one of the partitions and the (border-to-border) distance between these partitions is always considered for the lower-bound computation.

Table 2. Lower-bound Quality

Network	*Naive* (%)	*ALT* (%)	*DL* (%)
CA	21	42	63
LA	33	46	66

6.2.4 Bidirectional vs. Unidirectional Search

In another set of experiments, we study the impact of path length (i.e., distance from s to d) on the speed-up of bidirectional search. Hence, we measure the performance of B-TDFP and U-TDFP with respect to distance by varying the path distance (1 to 300 miles) between s and d. Figure 7(b) shows the speed-up with respect to distance. We observe that the speed-up is significantly more especially for long distance queries. The reason is that for short distances the computational overhead incurred by B-TDFP is not worthwhile as U-TDFP visits less number of nodes anyway.

7 Conclusion and Future Work

In this paper, we proposed a time-dependent fastest path algorithm based on bidirectional A*. Unlike the most path planning studies, we assume the edge weights of the

road network are time varying rather than constant. Therefore, our approach yield a much more realistic scenario, and hence, applicable to the to real-world road networks. We also compared our approaches with those handful of time-dependent fastest path studies. Our experiments with real-world road network and traffic data showed that our proposed approaches outperform the competitors in storage and response time significantly. We intend to pursue this study in two different directions. First, we plan to investigate new data models for effective representation of spatiotemporal road networks. This is critical in supporting development of efficient and accurate time-dependent algorithms, while minimizing the storage and computation costs. Second, to support rapid changes of the traffic patterns (that may happen in case of accidents/events, for example), we intend to study incremental update algorithms for both of our approaches.

References

1. Batz, G.V., Delling, D., Sanders, P., Vetter, C.: Time-dependent contraction hierarchies. In: ALENEX (2009)
2. Cooke, L., Halsey, E.: The shortest route through a network with timedependent internodal transit times. Journal of Mathematical Analysis and Applications (1966)
3. Dean, B.C.: Algorithms for min-cost paths in time-dependent networks with wait policies. Networks (2004)
4. Dehne, F., Omran, M.T., Sack, J.-R.: Shortest paths in time-dependent fifo networks using edge load forecasts. In: IWCTS (2009)
5. Delling, D.: Time-dependent SHARC-routing. In: Halperin, D., Mehlhorn, K. (eds.) Esa 2008. LNCS, vol. 5193, pp. 332–343. Springer, Heidelberg (2008)
6. Delling, D., Wagner, D.: Landmark-based routing in dynamic graphs. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 52–65. Springer, Heidelberg (2007)
7. Demiryurek, U., Kashani, F.B., Shahabi, C.: A case for time-dependent shortest path computation in spatial networks. In: ACM SIGSPATIAL (2010)
8. Demiryurek, U., Pan, B., Kashani, F.B., Shahabi, C.: Towards modeling the traffic data on road networks. In: SIGSPATIAL-IWCTS (2009)
9. Ding, B., Yu, J.X., Qin, L.: Finding time-dependent shortest paths over large graphs. In: EDBT (2008)
10. Dreyfus, S.E.: An appraisal of some shortest-path algorithms. Operations Research 17(3) (1969)
11. Foschini, L., Hershberger, J., Suri, S.: On the complexity of time-dependent shortest paths. In: SODA (2011)
12. George, B., Kim, S., Shekhar, S.: Spatio-temporal network databases and routing algorithms: A summary of results. In: Papadias, D., Zhang, D., Kollios, G. (eds.) SSTD 2007. LNCS, vol. 4605, pp. 460–477. Springer, Heidelberg (2007)
13. Goldberg, A.V., Harellson, C.: Computing the shortest path: A* search meets graph theory. In: SODA (2005)
14. Gonzalez, H., Han, J., Li, X., Myslinska, M., Sondag, J.P.: Adaptive fastest path computation on a road network: A traffic mining approach. In: VLDB (2007)
15. Guc, B., Ranganathan, A.: Real-time, scalable route planning using stream-processing infrastructure. In: ITS (2010)
16. Halpern, J.: Shortest route with time dependent length of edges and limited delay possibilities in nodes. Mathematical Methods of Operations Research (1969)
17. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics (1968)

18. Kanoulas, E., Du, Y., Xia, T., Zhang, D.: Finding fastest paths on a road network with speed patterns. In: ICDE (2006)
19. Kohler, E., Langkau, K., Skutella, M.: Time-expanded graphs for flow-dependent transit times. In: Proc. 10th Annual European Symposium on Algorithms (2002)
20. Kolahdouzan, M., Shahabi, C.: Voronoi-based k nearest neighbor search for spatial network databases. In: VLDB (2004)
21. Nannicini, G., Delling, D., Liberti, L., Schultes, D.: Bidirectional a* search for time-dependent fast paths. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 334–346. Springer, Heidelberg (2008)
22. NAVTEQ, `http://www.navteq.com` (accessed in May 2010)
23. Orda, A., Rom, R.: Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. J. ACM (1990)
24. PeMS, `https://pems.eecs.berkeley.edu` (accessed in May 2010)
25. Pohl, I.: Bi-directional search. In: Machine Intelligence. Edinburgh University Press, Edinburgh (1971)
26. Potamias, M., Bonchi, F., Castillo, C., Gionis, A.: Fast shortest path distance estimation in large networks. In: CIKM (2009)
27. Samet, H., Sankaranarayanan, J., Alborzi, H.: Scalable network distance browsing in spatial databases. In: SIGMOD (2008)
28. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 568–579. Springer, Heidelberg (2005)
29. Sanders, P., Schultes, D.: Engineering fast route planning algorithms. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 23–36. Springer, Heidelberg (2007)
30. TELEATLAS, `http://www.teleatlas.com` (accessed in May 2010)
31. Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 776–787. Springer, Heidelberg (2003)