Querying Shortest Paths on Time Dependent Road Networks

Yong Wang
Tsinghua University
wangy18@mails.tsinghua.edu.cn

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

Nan Tang
QCRI, HBKU
ntang@hbku.edu.qa

ABSTRACT

For real-world *time dependent road networks* (TDRNs), answering shortest path-based route queries and plans in real-time is highly desirable by many industrial applications. Unfortunately, traditional (*Dijkstra-* or A^*-like) algorithms are computationally expensive for such tasks on TDRNs.

Naturally, indexes are needed to meet the real-time constraint required by real applications. In this paper, we propose a novel height-balanced tree-structured index, called TD-G-tree, which supports fast route queries over TDRNs. The key idea is to use hierarchical graph partitioning to split a road network into hierarchical partitions. This will produce a balanced tree, where each tree node corresponds to a partition and each parent-child relationship corresponds to a partition and its sub-partition. We then compute and index *time dependent shortest paths* (TDSPs) only for *borders* (*i.e.,* vertices whose edges are cut by a partition). Based on TD-G-tree, we devise efficient algorithms to support TDSP queries, as well as time-interval based route planning, for computing optimal solutions through dynamic programming and chronological divide-and-conquer. Extensive experiments on real-world datasets show that our method significantly outperforms existing approaches.

1. INTRODUCTION

Route querying and planning over road networks are important in many applications, especially for GPS navigation services. Consider a group of tourists shopping in the Desert Outlets who want to join the Night Tour in Las Vegas Downtown. To avoid traffic jam and get more time for shopping and sightseeing, they want to find the optimal departure time, as well as minimizing the travel time on the way.

Real-world road networks are typically formalized as time-dependent road networks (TDRNs), where an edge is associated with a time dependent weight function for modeling time-dependent speed. Shortest path based queries over TDRNs raise several algorithmic challenges.

Non-index methods are typically *Dijkstra-* or A^*-like [4, 10, 21, 23, 30], which usually take dozens of seconds to answer a query over large TDRNs, thus cannot meet the real-time constraint. Index-based methods either suffer from unstable preprocessing time over time-varying edge weights (TCH [2]), or unstable performance caused by inaccurate cost estimation of A^*-based heuristics (TDCALT [7]).

On one side, non-index methods over large TDRNs are not efficient enough. On the other side, indexing all shortest paths for all time intervals is prohibitively expensive in space. Hence, our *key observation* is that, if we can smartly select only a small number of shortest paths to index, we may be able to efficiently answer different types of single-pair shortest path queries in TDRNs.

We propose a height-balanced tree index, *i.e.,* TD-G-tree, based on hierarchical graph partitioning over a TDRN. More specifically, each tree node corresponds a graph partition, and its children nodes are associated with corresponding sub-partitions. We maintain shortest paths between borders of partitions, where a *border* is a vertex with at least an edge being cut by a partition.

We also show that how TD-G-tree can support single-pair time dependent shortest path queries, including: time dependent shortest path (TDSP) that asks the best route at a specific time, and time-interval based planning (TIP) that looks for the best route within a specific time interval. The basic idea is that we use the borders to guide us to compute the time-dependent shortest paths and do not need to blindly traverse the graphs.

There are two requirements for TD-G-tree to be used by a navigation system. As usually traffic flow pattern of the same day of the weeks may not change dramatically, historical time-dependent road network data [12] is needed for constructing the TD-G-tree. Moreover, emergent traffic flow changes on some roads from the real-time traffic is supplemented to search alternative routing plans and update TD-G-tree periodically.

Our main contributions are summarized as follows.

1. We formalize TDRNs, introduce time dependent weight functions associated with edges, and define different types of queries studied in this work (Section 2).
2. We present a new index, TD-G-tree, and describe its construction and maintenance (Section 3).
3. We devise novel algorithms to efficiently support popular time-dependent queries over TDRNs, using TD-G-tree (Section 4).
4. We conduct experiments to show the superiority of TD-G-tree over existing approaches (Section 5).

[1]Guoliang Li is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342265

Table 1: Table of Notations

Notation	Definition
$G(V, E, W, T)$	A graph for a time dependent road network
$\omega_{v_i v_j}(t)$	Time-dependent weight function of edge $e_{v_i v_j}$
$\rho_{v_s v_e}$	A path from v_s to v_e: $\langle v_s = v_1, \cdots, v_k = v_e \rangle$
$\varrho_{v_s v_e}$	The set of all paths from v_s to v_e
$\tau_{\rho_{v_s v_e}}(t)$	Travel-time function from v_s to v_e, via path $\rho_{v_s v_e}$
$\tau^*_{v_s v_e}(t)$	Shortest travel time function from v_s to v_e
$B(G_i)$	The border set of G_i
M_i	Travel-time matrix of G_i
$X(G_i)$	The vertices in the travel-time matrix M_i
leaf(v)	The corresponding leaf node of vertex v
LCA(G_i, G_j)	The least common ancestor node of G_i and G_j

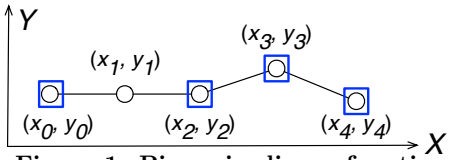


Figure 1: Piecewise linear function

2. BACKGROUND

2.1 Time Dependent Road Networks

We summarize the notations used in this paper in Table 1.

Definition 1. *[Time Dependent Directed Graph.] A time dependent road network is modeled as a directed graph* $G = (V, E, W, T)$*, where each* $v_i \in V$ *is a* vertex*,* $e_{v_i v_j} \in E$ *is a* directed edge *from* v_i *to* v_j*,* $\omega_{v_i v_j} \in W$ *is a* time dependent weight function *associated with edge* $e_{v_i v_j}$*, and* T *is a* time interval $[\mathbf{t_s}, \mathbf{t_e}]$.

We use **piecewise linear functions** (PLF) to model the edge weights. Given two points (x_0, y_0) and (x_1, y_1) where $x_0 < x_1$, the *linear interpolant* is the straight line between them: for any $x \in [x_0, x_1]$, its y is $f(x) = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}$. Given a set P of consecutive points $(x_0, y_0), \ldots, (x_m, y_m)$, its set of *interpolation points*, denoted by P^I, is a subset of P, by removing all points (x_i, y_i) $(0 < i < m)$ in P where $\frac{y_i - y_{i-1}}{x_i - x_{i-1}} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$. That is, the set P^I of interpolation points is sufficient to represent P, which can significantly save space overhead.

Example 1: [Piecewise Linear Functions.] Consider a set of points $\{(x_0, y_0), \ldots, (x_4, y_4)\}$ in Figure 1. Its set of *interpolation points* is $\{(x_0, y_0), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$, where (x_1, y_1) is removed. There are three *linear interpolants*, from (x_0, y_0) to (x_2, y_2), (x_2, y_2) to (x_3, y_3), and (x_3, y_3) to (x_4, y_4) – each linear interpolant corresponds to a linear function. □

Definition 2. *[Piecewise Linear Weight Functions.] Given an edge* e *associated with a set of weights (travel time) at specific time as* $S = \{(t_1, w_1), \ldots, (t_k, w_k)\}$ *where* $t_1 = \mathbf{t_s}$ *and* $t_k = \mathbf{t_e}$*, let* S^I *denote the set of interpolation points of* S*. The* piecewise linear weight function ω *for edge* e *is the linear interpolants for* S^I*, and the evaluation of the function at time* t*,* i.e., $\omega_{v_i v_j}(t)$*, returns a weight, indicating the* time *needed to move from* v_i *to* v_j *departing at time* t.

Computing Piecewise Linear Weight Functions. We first compute the average travel cost on each road in every 5 minutes using historical data. We then take the average cost as a discrete point, and use piecewise linear functions (PLF) to fit these discrete points [18]. Note that the weight function for edge $e_{v_i v_j}$ might be different from that for edge $e_{v_j v_i}$ in practice. In this paper, we simply assume they are the same for ease of presentation – treating them differently will not affect our algorithms.

Table 2: Edge Weights ($e_{v_i v_j} = e_{v_j v_i}$)

Edge	Weights {(time, weight)}
$e_{v_1 v_2}$	$\{(0, 8), (20, 8), (35, 20), (60, 20)\}$
$e_{v_0 v_2}$	$\{(0, 8), (60, 8)\}$
$e_{v_0 v_1}$	$\{(0, 4), (60, 4)\}$
$e_{v_1 v_9}$	$\{(0, 5), (60, 5)\}$
$e_{v_6 v_9}$	$\{(0, 10), (50, 15), (60, 15)\}$
$e_{v_2 v_3}$	$\{(0, 15), (60, 15)\}$
$e_{v_3 v_4}$	$\{(0, 6), (60, 6)\}$
$e_{v_3 v_6}$	$\{(0, 30), (30, 10), (60, 10)\}$
$e_{v_2 v_6}$	$\{(0, 5), (60, 5)\}$
$e_{v_9 v_{10}}$	$\{(0, 2), (60, 2)\}$
$e_{v_{10} v_{11}}$	$\{(0, 3), (60, 3)\}$

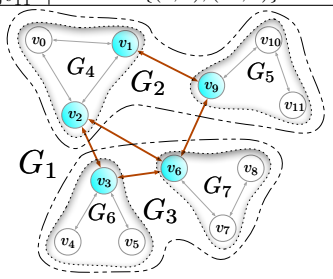
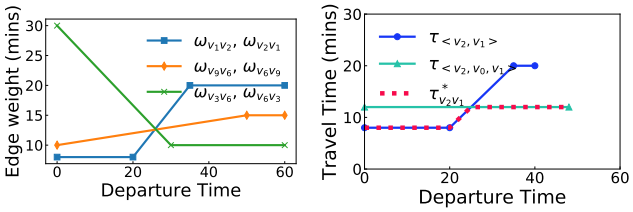


Figure 2: A sample graph for a TDRN

Example 2: [Graph.] Figure 2 shows a time dependent directed graph (please ignore the circles for now, which will be discussed in Section 3 for hierarchical graph partitioning).

[Edge Weights.] Edge weights are given in Table 2. For simplicity, we assume the weight for edge $e_{v_i v_j}$ is the same as $e_{v_j v_i}$, so only one is given. Each edge is associated with a set of (time, weight) pairs, *e.g.,* the weight $(0, 8)$ for $e_{v_1 v_2}$ says that it takes 8 minutes from v_1 to v_2 at time 0.

[Piecewise Linear Weight Functions.] Consider Figure 2 and its edge weights in Table 2. Some sample piecewise linear weight functions are shown in Figure 3(a), where edge weights are time-varying, such as $\omega_{v_2 v_1}(t)$. □



(a) Weight Functions (b) Travel-Time Functions

Figure 3: Sample PLFs

Definition 3. *[Path.] A* **path** $\rho_{v_s v_e}$ *is a sequence of connected vertices* $\langle v_s, \ldots, v_e \rangle$*. We write* $\varrho_{v_s v_e}$ *as the set of all paths from* v_s *to* v_e.

Definition 4. *[Travel Time Function of a Path.] Given a path* $\rho_{v_1 v_i}$*, let* $\rho_{v_1 v_{i-1}}$ *be a sub-path of* $\rho_{v_1 v_i}$ *by removing the last vertex* v_i*. The* travel time function *for path* $\rho_{v_1 v_i}$ *starting at time* t*, denoted by* $\tau_{\rho_{v_1 v_i}}(t)$*, is recursively defined as:* $\tau_{\rho_{v_1 v_i}}(t) = \tau_{\rho_{v_1 v_{i-1}}}(t) + \omega_{v_{i-1} v_i}(t + \tau_{\rho_{v_1 v_{i-1}}}(t))$*, which is the time needed from* v_1 *to* v_{i-1} *via path* $\rho_{v_1 v_{i-1}}$ *departing at* t*, plus the time from* v_{i-1} *to* v_i *after reaching* v_{i-1}*. Obviously,* $\tau_{\rho_{v_1 v_1}}(t) = 0$.

Now we introduce the travel time function of the shortest path between two vertices departing at a specified time.

Definition 5. *[Shortest Travel Time Function.] The* shortest travel time function *(STTF) from* v_s *to* v_e *at time* t*, denote by* $\tau^*_{v_s v_e}(t)$*, is the* minimum *time from all possible paths* $\varrho_{v_s v_e}$*,* i.e., $\tau^*_{v_s v_e}(t) = \min_{\rho_{v_s v_e}} \{\tau_{\rho_{v_s v_e}}(t) \mid \rho_{v_s v_e} \in \varrho_{v_s v_e}\}$*. The corresponding shortest path is denoted by* $\rho^*_{v_s v_e}(t)$.

Note that we implicitly assume the FIFO [23] property: for any time $t \leq t'$, we have $t + \tau^*_{v_s v_e}(t) \leq t' + \tau^*_{v_s v_e}(t')$.

Example 3: [Paths.] This, $\langle v_0, v_1, v_9 \rangle$, is a path from v_0 to v_9. $\langle v_0, v_2 \rangle, \langle v_0, v_1, v_2 \rangle$ and $\langle v_0, v_1, v_9, v_6, v_2 \rangle$ are in $\varrho_{v_0 v_2}$.

[Travel Time Functions.] Some sample travel time functions are given in Figure 3(b). For instance, $\tau_{\langle v_2, v_1 \rangle}(t) = \omega_{v_2 v_1}(t)$ and $\tau_{\langle v_2, v_0, v_1 \rangle}(t) = \omega_{v_2 v_0}(t) + \omega_{v_0 v_1}(t + \omega_{v_2 v_0}(t))$.

[Shortest Travel Time Functions.] One shortest travel time function is $\tau^*_{v_2 v_1}(t) = \min_{\rho_{v_2 v_1}} \{\tau_{\rho_{v_2 v_1}}(t) \mid \rho_{v_2 v_1} \in \varrho_{v_2 v_1}\}$, which is shown as the bold dashed line in Figure 3(b). □

2.2 Time Dependent Shortest Path Queries

We introduce two types of widely used queries.

DEFINITION 6. *[Time Dependent Shortest Path (TDSP).] Given a source vertex v_s, a target vertex v_e, and a departure time t from v_s, the* time dependent shortest path *query, denoted by $Q(v_s, v_e, t)$, returns the exact fastest travel time $\tau^*_{v_s v_e}(t)$ and its associated path $\rho^*_{v_s v_e}(t)$.*

The other type is for route planning – one always wants to find the fastest route from one vertex to another, departing within a specific time interval, instead of just a time point.

DEFINITION 7. *[Time Interval Planning (TIP).] Given a source vertex v_s, a target vertex v_e, and a departure time interval $I = [t, t']$, the* time interval planning *query, denoted by $Q(v_s, v_e, I)$, returns $t^* \in I$, at which it takes the exact fastest time from v_s to v_e, denoted by $\tau^*_{v_s v_e}(t^*)$, and its associated path $\rho^*_{v_s v_e}(t^*)$.*

Note that, for simplifying our discussion, we focus on *departure time* interval when using I. However, it could also be *arrival time interval* for a different type of route planning.

Example 4: Consider Figure 2, 3(b) and Table 2. Let's discuss different types of queries from v_2 to v_1.

[TDSP: $Q(v_2, v_1, 10)$.] It has a fixed departure time $t = 10$. A TDSP algorithm will return the fastest travel time (8 mins) through optimal path $\langle v_2, v_1 \rangle$.

[TIP: $Q(v_2, v_1, [20, 60])$.] It has a time interval $[20, 60]$. A TIP algorithm will return the optimal departure time as $t^* = 20$ with fastest travel time (8 mins), and the associated optimal path is $\langle v_2, v_1 \rangle$. □

3. TD-G-TREE: TIME DEPENDENT INDEX

As mentioned earlier, shortest path related queries over large TDRNs require high computational cost, such that traditional algorithms are not efficient enough to support real-world applications. Apparently, indexes are needed.

Our key idea is inspired from two aspects: (1) Hierarchical partitions are widely used to index spatial data, such as R-tree. (2) No one can afford to index all pairs of shortest paths – with time dependent edge functions the number will be blown up. Fortunately, oftentimes, only a small number of time dependent shortest paths are needed to efficiently support shortest paths of all pairs, such as G-tree [38] that indexes static shortest paths based on partitions.

In this section, we will first define our proposed index TD-G-tree in Section 3.1, and then elaborate how to construct and update it in Section 3.2 and 3.3 respectively.

3.1 The TD-G-Tree Index

Before defining TD-G-tree, let's introduce its main concepts: (hierarchical) *graph partitioning*, *border vertices*, and *travel-time matrix* for each partition.

DEFINITION 8. *[Graph Partitioning.] Given a graph $G = (V, E, W, T)$, a* graph partitioning *of G, denoted by $\Delta(G)$, is a set of n subgraphs $G_i = (V_i, E_i, W_i, T)$ ($i \in [1, n]$), where (1) $V = \bigcup_{i \in [1,n]} V_i$; (2) $V_i \cap V_j = \emptyset$ if $i \neq j$; and (3) each G_i is an* induced subgraph *of G* – i.e., *consisting of vertices V_i and all edges connecting pairs of vertices V_i in E and their associated weight functions.*

Example 5: [Graph Partitioning.] Figure 2 shows sample partitions, *e.g.,* $\Delta(G_1) = \{G_2, G_3\}$ is a graph partitioning over G_1.

[Hierarchical Graph Partitioning.] We can further partition G_2 into $\{G_4, G_5\}$, and G_3 into $\{G_6, G_7\}$, which are also shown in Figure 2. □

We will discuss how to perform hierarchical graph partitioning in Section 3.2. Note that during graph partitioning, all edges across partitions are cut, where corresponding vertices are called border vertices (or simply as *borders*). Borders are important for our index as we can safely hop over non-border vertices through borders during query processing. Formally, we define borders below.

DEFINITION 9. *[Border.] Given a graph $G = (V, E, W, T)$, let $G_x(V_x, E_x, W_x, T)$ be one of its subgraphs generated by hierarchical graph partitioning of G. A vertex $v_i \in V_x$ is a* border *of G_x if there exists an edge $e_{v_i v_j} \in E$ but $e_{v_i v_j} \notin E_x$. We use $B(G_x)$ to denote the set of borders of graph G_x.*

Example 6: [Borders.] Consider sub-partitions $\Delta(G_1) = \{G_2, G_3\}$ of G_1 in Figure 2. We have $B(G_2) = \{v_2, v_9\}$ and $B(G_3) = \{v_3, v_6\}$. Recursively, we get $B(G_4) = \{v_1, v_2\}$. □

DEFINITION 10. *[Travel-Time Matrix.] Given a set X of vertices, a* travel-time matrix *M maintains shortest travel time functions (see Definition 5) of all pairs of vertices in X. Given $\forall v_i, v_j \in X$ and a departure time t, $M[v_i, v_j](t)$ returns $\tau^*_{v_i v_j}(t)$, the shortest time from v_i to v_j.*

We simply write *matrix* for travel-time matrix. Intuitively, we want to maintain a matrix for each partition, such that any route query can be efficiently computed by using these matrices. Because the vertices of a sub-partition (*e.g.,* $\{v_0, v_1, v_2\}$ of G_4) are clearly also vertices of its parent partition (*i.e.,* G_2), we need to be careful to avoid data redundancy, for which we differentiate two types of partitions:

- *Leaf Partitions:* those partitions without sub-partitions, such as $\{G_4, G_5, G_6, G_7\}$.
- *Internal Partitions:* those with sub-partitions, such as $\{G_1, G_2, G_3\}$.

Intuitively, for a leaf partition G_L, all vertices need to be indexed in M_L, *i.e.,* $X(G_L) = V_L$. However, we do not maintain matrix entires between non-border vertex pairs to save its huge space cost. As will be shown later in Section 4, we can recover shortest paths even without these pairs. For an internal partition G_I, the vertices needed for its matrix are the union of all its sub-partitions' borders, $X(G_I) =$

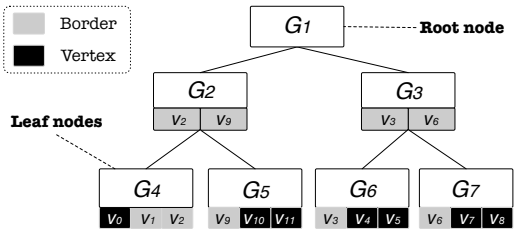


Figure 4: A sample TD-G-tree

$\cup_{G_c \in \Delta(G_I)} B(G_c)$, because all other path information within G_I can be assembled by the matrices of its sub-partitions.

Example 7: [Leaf Partitions.] We have $X(G_4) = \{v_0, v_1, v_2\}$ and $X(G_6) = \{v_3, v_4, v_5\}$.

[Interval Partitions.] We have $X(G_2) = \{v_1, v_2, v_9\}$, because $\Delta(G_2) = \{G_4, G_5\}$, $B(G_4) = \{v_1, v_2\}$ and $B(G_5) = \{v_9\}$. Similarly, we have $X(G_3) = \{v_3, v_6\}$. Moreover, $X(G_1) = \{v_2, v_3, v_6, v_9\}$, because $\Delta(G_1) = \{G_2, G_3\}$, $B(G_2) = \{v_2, v_9\}$ and $B(G_3) = \{v_3, v_6\}$.

Also, note that $B(G_2) = \{v_2, v_9\}$ (Figure 4), and it is not the same as $X(G_2) = \{v_1, v_2, v_9\}$, which is the union of borders of G_2's child-nodes, $B(G_4)$ and $B(G_5)$.

[Travel-time Matrices.] Figure 5 shows two matrices: M_3 for an internal partition G_3, and M_4 for a leaf partition G_4. In M_4, v_0 is a non-border vertex, and v_1, v_2 are borders of G_4. Specifically, entry $M_4[v_2, v_1]$ represents the shortest travel time function from v_2 to v_1 ($\tau^*_{v_2 v_1}$ in Figure 3(b)). □

We are now ready to define our proposed index. Typically, any hierarchical partitioning needs two parameters: one to restrict the number of partitions for each level of partitioning, and the other to constrain the size of a leaf partition.

`TD-G-tree`. Given a `TDRN` graph G_1, fanout parameter κ_f, and the maximum number of vertices in a leaf node κ_l, its `TD-G-tree` is a height-balanced tree that:

1. Each tree node n_i represents a partition G_i, $i \in [1, m]$, where the m partitions are generated by hierarchical graph partitioning. The root tree node n_1 is for G_1.
2. Each internal node has κ_f (≥ 2) child nodes.
3. Each leaf node contains at most κ_l (≥ 1) graph vertices, and all leaf nodes appear at the same level.
4. Each node n_i maintains a travel-time matrix for G_i.

Properties (1)-(3) ensure that `TD-G-tree` is a balanced tree; (4) is used to accelerate the performance of time dependent shortest path search (see more details in Section 4).

For convenience, we use *"vertex"* for a vertex in the road-network graph, and *"node"* for a node in the `TD-G-tree`. Also, we will use the tree node n_i and its corresponding partition G_i interchangeably.

Example 8: [`TD-G-tree`.] Figure 4 shows a `TD-G-tree` for the `TDRN` in Figure 2. The tree nodes are generated by hierarchical graph partitioning (*e.g.,* Example 5). For an internal partition such as G_3, we maintain a matrix (Figure 5(a)) for its $X(G_3) = \{v_3, v_6\}$. For leaf partition such as G_4, we maintain a matrix (Figure 5(b)) for all its vertices $V_3 = \{v_0, v_1, v_2\}$. □

3.2 TD-G-Tree Construction

The construction of a `TD-G-tree` for a `TDRN` G consists of two parts: its tree structure and the associated matrix for each tree node. The tree structure is computed based on the topology of G, and the matrix of each tree node is computed based on vertices $X(G_i)$ associated with each tree node G_i.

M_3	v_3	v_6
v_3	$\{(0,0),(60,0)\}$	$\{(0,20),(15,20),(30,10),(50,10)\}$
v_6	$\{(0,20),(15,20),(30,10),(50,10)\}$	$\{(0,0),(60,0)\}$

(a) Matrix M_3 of sub-graph G_3

M_4	v_0	v_1	v_2
v_0	———	$\{(0,4),(56,4)\}$	$\{(0,8),(52,8)\}$
v_1	$\{(0,4),(56,4)\}$	$\{(0,0),(60,0)\}$	$\{(0,8),(20,8),(25,12),(48,12)\}$
v_2	$\{(0,8),(52,8)\}$	$\{(0,8),(20,8),(25,12),(48,12)\}$	$\{(0,0),(60,0)\}$

(b) Matrix M_4 of sub-graph G_4

Figure 5: Travel time matrices for G_3 and G_4

3.2.1 Computing TD-G-Tree Nodes based on Hierarchical Graph Partitioning

Given a `TDRN` graph G, a fanout parameter κ_f, and a leaf node size threshold κ_l, it first partitions G into κ_f approximately equal-sized subgraphs (*i.e.,* $|V_1| \approx \cdots \approx |V_{\kappa_f}|$), which form the child-nodes of G, *i.e.,* $\{n_1, \ldots, n_{\kappa_f}\}$. It then recursively partitions each of the above subgraphs n_i ($i \in [1, \kappa_f]$). The entire process terminates when the new subgraphs have at most κ_l vertices, *i.e.,* reaching the leaf tree nodes.

Example 9: Figure 2 shows a hierarchical graph partitioning of G_1, by setting $\kappa_f = 2$ and $\kappa_l = 4$. □

Note that finding the optimal graph partitioning for minimizing cut edges is NP-hard [22]. In this paper, we adopt a well-known multilevel partitioning algorithm [22], which minimizes the number of edges to cut, and guarantees that each subgraph has nearly the same size through a balancing factor. As a result, we get a balanced `TD-G-tree`.

3.2.2 Computing Matrices for Tree Nodes

The primary goal of using matrices is to maintain shortest travel time functions to accelerate shortest path queries $Q(v_s, v_e)$ between two vertices, *i.e.,* by *only* visiting a small part of borders in least common tree node n that contains both vertices v_s and v_e, and descendent tree nodes of n.

We face two main challenges for computing matrices.

[**C1.**] *Redundant Matrix Entries.* For any tree node n_i (for partition G_i), all vertices of G_i also belong to the parent-node n_j of n_i, *i.e.,* G_i is subgraph of G_j. Evidently, maintaining all graph vertices in the matrix for each partition will cause heavy data redundancy.

[**C2.**] *Expensive Computation.* For any two vertices in a matrix, we need to compute the global optimal travel time functions between them, which oftentimes involves vertices that are not in the same partition – running a global algorithm to compute shortest travel time functions (`STTF`*s*) for each tree node is prohibitively expensive.

To tackle challenge C1, we have discussed earlier the graph vertices that are needed to maintain for each matrix (see Example 7). Besides, the entries of border pairs shared by different matrices refer to the same physical memory, which eliminates redundancy caused by shared borders. For challenge C2, we propose a novel two-phase algorithm: we first compute the "local optimal" travel-time functions in a bottom-up fashion over the `TD-G-tree`, and then compute the "global optimal" travel-time function in a top-down fashion to gradually refine each imprecise matrix entry. Next, let's describe our proposed two-phase approach.

Two-Phase Algorithm. The two-phase method – "approximate: bottom-up local-optimal" and "precise: top-

Algorithm 1: Two-Phase Matrix Building

```
   Input: M = M_1, ..., M_k : Matrices, W: Edge weights
   Output: M: Matrices with optimal entries
   //  Approximate: Bottom-up Local-optimal
1  Initialize matrices entries with W;
2  foreach node n_i from i = k to 1 do
3      TD-Floyd (M_i);
4      if n_i ≠ root, let n_f the parent node of n_i then
5          foreach two distinct vertices
             v_s, v_e ∈ X(G_i) ∩ X(G_f) do
6            M_f[v_s, v_e] = M_i[v_s, v_e];
   //  Precise: Top-Down Global-optimal
7  foreach non-leaf node n_i from i = 1 to k do
8      foreach child node G_c of G_i do
9          foreach two distinct vertices
             v_s, v_e ∈ X(G_c) ∩ X(G_i) do
10           if M_i[v_s, v_e] ≠ M_c[v_s, v_e] then
11             M_c[v_s, v_e] = M_i[v_s, v_e];
12             Mark M_c as dirty;
13       if M_c is dirty then
14         TD-Floyd (M_c);
```

Algorithm 2: TD-Floyd (M)

```
   Input: M: A travel time matrix for partition G
   Output: M: A refined travel time matrix
1  foreach vertex v_k in the vertices X(G) of matrix M do
2      foreach v_s in X(G) do
3          foreach v_e in X(G) do
4            τ'_{v_s v_e} = M[v_s, v_k] + Com(M[v_k, v_e], M[v_s, v_k]);
5            M[v_s, v_e] = Min(M[v_s, v_e], τ'_{v_s v_e});
```

down global-optimal" – to compute the travel-time functions for each matrix, is shown in Algorithm 1.

To summarize, during the bottom-up local-optimal phase (lines 2–6), we compute the matrix of a partition G_i by *only* considering the information within G_i. This phase is *approximate* (local optimal) because the precise STTF may involve paths via some vertices outside of G_i. In the top-down phase (lines 7–14), we use the global optimal information computed in G_1 (the entire graph) to refine those approximate matrix entries to be precise.

Approximate: Bottom-Up Local-Optimal

It works as follows, from leaf nodes (*i.e.,* bottom), to the internal nodes (*i.e.,* up), until the root (the loop in lines 2–6 is from k down to 1, *i.e.,* leaf to root).

For each tree node n_i, we first run a generalized Floyd algorithm (line 3, will be explained shortly) over only the vertices $X(G_i)$ in subgraph G_i, and the number of vertices in $X(G_i)$ is much smaller than the number of vertices in G_i.

Afterwards, a key *optimization* to share computation is that when a child-node G_i and its parent-node G_f share common vertices in their matrices, *i.e.,* those vertices in $X(G_i) \cap X(G_f)$, the corresponding matrix entry in M_f can be directly assigned (lines 5–6). A further *optimization* to save space is that instead of synchronizing $M_x[v_a, v_b]$ and $M_y[v_a, v_b]$ across different matrices, they directly use the same physical entry linked by pointers – a synchronization is then done automatically.

Precise: Top-Down Global-Optimal

After the above bottom-up phase, each matrix maintains the local optimal results for the corresponding tree node. Moreover, since G_1 is the entire graph, its local optimal is actually global optimal (the proof is omitted due to space limitation). In what follows, we discuss how to refine those local optimal results to become the global optimum.

For each non-leaf node starts from the root node (line 7), it checks the shared vertices between matrix composing set X of the current node and each of its child nodes (lines 8–9). If it identifies that the shared vertices' entries in the child-node matrix are not global optimal (line 10), it will first use the guaranteed global optimal results from current node to update this child-node's matrix (line 11), and mark it as dirty (line 12). After finishing checking shared vertices' for a child-node matrix M_c, if M_c is updated, we re-run the TD-Floyd algorithm on M_c to achieve global optimal (lines 13-14). Otherwise, M_c and its decedent matrices are already optimal after the Approximate: Bottom-up Local-optimal phase, and we don't need to recompute them.

The TD-Floyd *Algorithm*

Next, let's complete the discussion of Algorithm 1 by describing the TD-Floyd algorithm used in lines 3 and 14 of Algorithm 1. However, before presenting it, let's pause and introduce several important functions.

Com($\tau_{v_s v_k}$, $\tau_{v_k v_e}$). Given two travel-time functions $\tau_{v_s v_k}$ and $\tau_{v_k v_e}$, the Com() operation refines the travel-time function $\tau_{v_k v_e}$ by *compounding* the departure time from t to $\tau_{v_s v_k}(t)$; that is, $\tau_{v_k v_e}(t + \tau_{v_s v_k}(t))$.

Min($\tau_{v_s v_e}, \tau'_{v_s v_e}$). Given two travel-time functions $\tau_{v_s v_e}, \tau'_{v_s v_e}$, the Min() operation returns a new travel-time function with shorter travel time for every departure time t (*i.e.,* a series of interpolation points), denoted by $\min\{\tau_{v_s v_e}, \tau'_{v_s v_e}\}$.

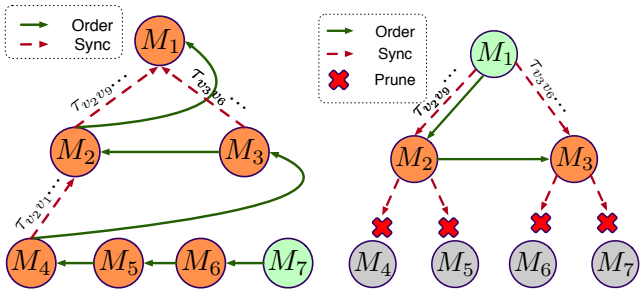
Example 10: [Com()] In Table 2, $\tau_{\langle v_0, v_2\rangle} = \omega_{v_0 v_2} = \{(0, 8), (52, 8)\}$. $\tau_{\langle v_2, v_1\rangle} = \omega_{v_2 v_1} = \{(0, 8), (20, 8), (35, 20), (40, 20)\}$. We get $\tau_{\langle v_2, v_1\rangle}(t + \tau_{\langle v_0, v_2\rangle}(t)) = \{(0, 8), (12, 8), (27, 20), (32, 20)\}$.

[Min()] In Figure 3(b), based on $\tau_{\langle v_2, v_0, v_1\rangle} = \{(0, 12), (48, 12)\}$ and $\tau_{\langle v_2, v_1\rangle} = \{(0, 8), (20, 8), (35, 20), (40, 20)\}$, we get $\tau^*_{v_2 v_1} = \{(0, 8), (20, 8), (25, 12), (48, 12)\}$ (other paths with larger travel time are omitted). □

TD-Floyd Algorithm. The TD-Floyd algorithm is given in Algorithm 2. It takes a matrix M as input, relaxes all entries and outputs the refined matrix through the dynamic programming scheme. More specifically, in each inner iteration (lines 3-5), we relax travel time function $\tau_{v_s v_e}$ with $\tau_{v_s v_k}$ and $\tau_{v_k v_e}$, where we may get part (or whole) of $\tau_{v_s v_e}(t), t \in T$ updated with shorter travel time and get a new function.

Example 11: [TD-G-tree Construction.] We show how to build travel-time matrices for TD-G-tree instance in Figure 4, with the entire process depicted in Figure 6. The partitions $M_1, \ldots, M_7$ are initialized with connected edges.

[Bottom-up.] We use TD-Floyd to calculate matrices one-by-one from leaf-node level to root node (solid line order in Figure 6(a)). In particular, we initialize $M_4[v_2, v_1] = \omega_{v_2 v_1}$ and relaxing via path $\langle v_2, v_0, v_1\rangle$ we get $M_4[v_2, v_1] = \tau^*_{v_2 v_1}$, because there is no faster path via vertices outside G_4. After the computation of M_4, because the border pair v_2 and v_1 is in $B(G_2)$, so we synchronize $M_4[v_2, v_1]$ to $M_2[v_2, v_1]$ (dashed line in Figure 6(a)). Similarly, during calculating M_3, we only get a viable path $\langle v_3, v_6\rangle$ for $M_3[v_3, v_6]$ and synchronize to $M_1[v_3, v_6]$. Another path $\rho_1 = \langle v_3, v_2, v_6\rangle$ is ignored during calculating M_3 since $v_2 \notin V_3$. ρ_1 is a faster



(a) Bottom-up local-optimal (b) Top-down global-optimal

Figure 6: Two-phase matrix building process

path than $\langle v_3, v_6\rangle$ during $t \in [0, 15]$, which will be considered during calculating $M_1[v_3, v_6]$.

[Top-down.] Next, we refine matrices in a top-down order from M_1 to M_7, where matrices already being global optimal after Approximate: Bottom-up Local-optimal will be pruned, *e.g.,* M_2's descendant-node matrices in Figure 6(b). In particular, we synchronize $M_1[v_3, v_6](t) = \tau^*_{v_3v_6}(t)$, $t \in [0, 15]$ (via shorter path $\langle v_3, v_2, v_6\rangle$)to $M_3[v_3, v_6]$, and mark M_3 as dirty. After processing M_1, because M_2 and M_3 are dirty, we re-calculate them and ensure their global optimality. □

Space Complexity. A tree node on the i-th level contains $O(|V|/\kappa_f^{i-1})$ vertices, where graph partitioning produces $O(\log_2 \kappa_f \cdot \sqrt{|V|/\kappa_f^{i-1}})$ borders (Planar Separator Theorem [38]). Thus the matrices size at level i is $O(\kappa_f^{i-1} \cdot (\log_2 \kappa_f \cdot \sqrt{|V|/\kappa_f^{i-1}})^2) = O(\log_2^2 \kappa_f \cdot |V|)$. We have $O(\frac{|V|}{\kappa_l})$ leaf nodes, and the tree height is $O(\log_{\kappa_f} \frac{|V|}{\kappa_l})$. Suppose a PLF on time domain T has $\alpha(T)$ linear pieces. The space complexity is $O(\log_{\kappa_f} \frac{|V|}{\kappa_l} \cdot \log_2^2 \kappa_f \cdot |V| \cdot \alpha(T))$.

3.3 TD-G-Tree Update

Below we describe how to update the index according to real-time traffics. However, time-dependent edge-weight update is really hard, since any edge-weight update may influence the path selection between any pair of vertices throughout the graph. Besides, edge-weight update may happen simultaneously to many edges that are randomly distributed on the graph, and we need to design a high-efficient update policy to avoid lots of redundant calculations. In all existing works for time-dependent road networks, only TDCALT [7] discussed the edge weight update problems. However, TDCALT considered a sub-case that only considers edge-weight increase circumstances. In this paper, we propose an update strategy with feasible time overheads to handle real-time multiple edge updates for any PLF forms.

Consider a single edge-weight update, which influences matrices entries initialized with it or exists shortest path via it. Hence, we locate all matrices influenced by the update and refine them to optimum. If there are multiple-edge updates, naturally, there may exist matrices influenced by more than one edge update.

TD-G-tree Update Algorithm. The basic idea of multiple edge-weight update algorithm, as shown in Algorithm 3, is a two-phase processing: Approximate: Bottom-up Local-optimal (lines 5–14) and Precise: Top-Down Global-optimal (lines 15–25). To avoid redundant calculations of the same matrix, we maintain two priority queues of node indices (lines 1–2), i.e., one for bottom-up phase (called UQ) and

Algorithm 3: Edge Weights Update

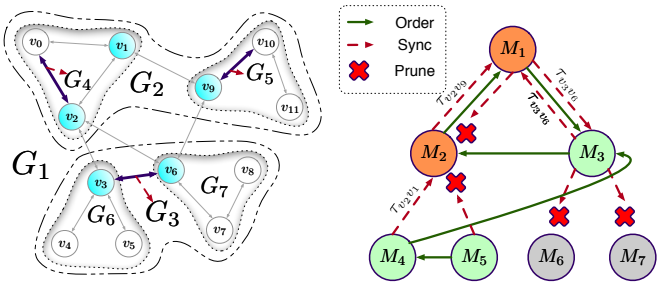
Input: Matrices M=$M_1, \cdots, M_n$, U: Updated weight set
Output: M: Matrices with optimal entries

```
1  Initialize greater first element-unique priority queue UQ = ∅;
2  Initialize less first element-unique priority queue DQ = ∅;
3  Re-initialize matrices entries with U;
4  foreach ω_{v_i v_j} ∈ U do UQ.push(LCA(leaf(v_i), leaf(v_j))); ;
5  while UQ ≠ ∅ do
6      n_i = UQ.popfront();
7      TD-Floyd (M_i);
8      if n_i ≠ root, let n_f the parent node of n_i then
9          foreach two distinct vertices
               v_s, v_e ∈ X(G_i) ∩ X(G_f) do
10             M_f[v_s, v_e] = M_i[v_s, v_e];
11         if M_f is updated then
12             UQ.push(n_f);
13             DQ.push(n_f);
14     else DQ.push(n_i); ;
15 while DQ ≠ ∅ do
16     n_i = DQ.popfront();
17     if n_i is a non-leaf node then
18         foreach child-node G_c of G_i do
19             foreach two distinct vertices
                   v_s, v_e ∈ X(G_c) ∩ X(G_i) do
20                 if M_i[v_s, v_e] ≠ M_c[v_s, v_e] then
21                     M_c[v_s, v_e] = M_i[v_s, v_e];
22                     Mark M_c as dirty;
23             if M_c is dirty then
24                 TD-Floyd (M_c);
25                 DQ.push(M_c)
```

the other for top-down phase (called DQ). Firstly, we locate matrices influenced by updated edges, re-initialize corresponding entries and enqueue UQ with associated nodes (lines 3–4). Then we dequeue node n_i with largest index from UQ (lines 5–6, i.e., node in higher level) and refine it with TD-Floyd (lines 7). If there are changes on matrix entries shared by M_i's father matrix M_f, we synchronize changes to M_f and enqueue it to UQ (line 9–12). Besides, when M_f is updated, we enqueue n_f to DQ for Precise: Top-Down Global-optimal (lines 13–14). During top-down phase, we process the node n_i with smallest index first (lines 15-17, i.e., node in lower level). Similarly, if there exists any changes to matrix entries shared by child nodes' matrices, synchronize changes to child matrices, refine them to global optimum and enqueue them to DQ (lines 18-25).

It is worth noting that we don't need to update matrices on the whole time domain. To get the influenced time domain, suppose the minimum time domain cover all edges update intervals to current matrix M_i is $[t_{\min}, t_{\max}]$ where $t_{\min}$ corresponds to update (v_s, v_e) with time domain $[t_{\min}, t_i]$ (or a series of edges). We calculate the earliest departure time $t'_{\min}$ from any vertex in M_i which can arrive v_s at $t_{\min}$. The influenced time domain of M_i is $[t'_{\min}, t_{\max}]$.

Example 12: [TD-G-tree Update.] Based on the TD-G-tree instance from Figure 4, we update some edges weights in Table 2, *i.e.,* $\omega_{v_{10}v_9}(t) = 5, t \in [30, 45]$ with $\mathsf{LCA}(v_{10}, v_9) = G_5$, $\omega_{v_2v_0}(t) = 0.8(t - 40) + 8, t \in [40, 60]$ with $\mathsf{LCA}(v_2, v_0) = G_4$, and $\omega_{v_3v_6}(t) = 12, t \in [30, 50]$ with $\mathsf{LCA}(v_3, v_6) = G_3$ as shown in Figure 7(a). For $\omega_{v_{10}v_9}$, the update only influences M_5. For $\omega_{v_2v_0}$, the update starts from M_4, then updates $M_4[v_2, v_1]$ to M_2 and re-calculates M_2. $M_2[v_2, v_9]$ is updated and we synchronize to M_1 and re-calculate M_1. There is no update to push down, thus it terminates.



(a) LCA of updated edges (b) Update multiple edges

Figure 7: Edge-weights update

For $\omega_{v_3v_6}$, a similar update starts from M_3. Then, after synchronization to M_1, $M_1[v_3, v_6]$ is updated in M_1, thus we synchronize it to $M_3[v_3, v_6]$ and re-calculate M_3. Altogether, referring to Figure 7(b), for multiple update of these three edges, we start from M_5, and then go to M_4, then to M_3, M_2, finally to M_1. Checking shared entries of M_1 and synchronizing changed ones, we find that only M_3 needs to be refined. After refining M_3, no updates synchronize to M_6 and M_7, the algorithm terminates. For calculation of matrix update time domain, let's consider update M_5 based on $\omega_{v_{10}v_9}(t), t \in [30, 45]$ as an example. $M_5[v_{10}, v_9]$ is influenced on $[30, 45]$ and $M_5[v_{11}, v_9]$ on $[27, 42]$, thus the update time domain for M_5 is $[27, 45]$. □

Algorithm 4: TDSP

Input: $Q = (v_s, v_e, t)$, and a TD-G-tree **T**
Output: Shortest travel time, and *incomplete* optimal path

1 Locate leaf(v_s) and leaf(v_e) by a hash table;
2 **if** *v_s and v_e are in the same leaf node G_L* **then**
3 $\quad \tau^*_{v_s v_e}(t) = M_L[v_s, v_e](t)$;
4 **else**
5 $\quad G_1$=leaf(v_s), G_k=leaf(v_e), G_A = LCA(G_1, G_k) ;
6 $\quad$ Find the tree path $G_1 \cdots G_A \cdots G_k$ in **T**;
7 $\quad$ Initialize $\{\tau^*_{v_s v_b}(t) = M_1[v_s, v_b](t) \mid v_b \in B(G_1)\}$;
8 $\quad$ **for** $j = 1$ *to* $k-1$ **do**
9 $\quad\quad$ **foreach** $v_b \in X(G_{j+1}) \setminus X(G_j)$ **do**
10 $\quad\quad\quad X = X(G_j) \cap X(G_{j+1})$;
11 $\quad\quad\quad \tau^*_{v_s v_b}(t) = \min_{v_x \in X}\{\tau^*_{v_s v_x}(t) + M_{j+1}[v_x, v_b](t + \tau^*_{v_s v_x}(t))\}$
12 $\quad \tau^*_{v_s v_e}(t) = \min_{v_b \in B(G_k)}\{\tau^*_{v_s v_b}(t) + M_k[v_b, v_e](t + \tau^*_{v_s v_b}(t))\}$

4. ANSWERING SHORTEST PATH QUERIES

In this section, we will study how to use the TD-G-tree for route queries by first describing algorithms for supporting TDSP (Section 4.1) and TIP (Section 4.2) queries.

As will be seen later, the above algorithms for TDSP and TIP will return the shortest time, but only with *incomplete* optimal path, such as $\langle v_4, v_3, v_2, v_9, v_{11}\rangle$ (see the red path in Figure 8(a)), while the *complete* path is $\langle v_4, v_3, v_2, v_0, v_1, v_9, v_{10}, v_{11}\rangle$. Hence, we will also present techniques for *path recovery* to get the complete path (Section 4.3).

4.1 TDSP Query

Before introducing the algorithm for TDSP, let's describe an important operation on travel time functions.

Evaluation(f, t). Given a travel time function f and a certain time point $t \in T$, the function returns the time needed on function f at time t. It works as follows. First, it retrieves the linear interpolant f_i for time t. The result is then calculated as $f(t) = f_i(t) = \frac{f(t_{i+1}) - f(t_i)}{t_{i+1} - t_i}(t - t_i) + f(t_i)$ (see more details in Section 2.1). Suppose the number of interpolation points of f is $O(\alpha(T))$, which is linear proportional to T. By using binary search, the evaluation time overhead is $O(\log_2 \alpha(T))$.

TDSP Algorithm. The algorithm for processing a TDSP query $Q = (v_s, v_e, t)$ is shown in Algorithm 4, which returns the shortest travel time from v_s to v_e when departing at t, and the associated *incomplete* optimal path.

Generally, we categorize TDSP queries into two cases.

(1) v_s and v_e are in the same leaf node. Let G_L be the subgraph *w.r.t.* this leaf node. Then the shortest travel time is simply to use the travel-time function at the matrix cell $M_L[v_s, v_e]$, *i.e.,* $Q(v_s, v_e, t) = M_L[v_s, v_e](t)$ (lines 2–3), by using the Evaluation(f, t) function. Note that when both v_s and v_e are non-border vertices of G_L, we adopt local TD-Dijkstra search to handle it.

(2) v_s and v_e are in different leaf nodes. Let G_1 and G_k be different leaf nodes that contain v_s and v_e, respectively. The basic idea is to search TDSP along the tree-node path from G_1 to G_k, and to iteratively revise the travel time along with the path. More concretely, it works as follows.

[Locating Ancestor.] It finds the least common ancestor (LCA) of G_1 and G_k, denoted by G_A. This forms a unique tree path $G_1, \ldots, G_A, \ldots G_k$ (lines 5–6).

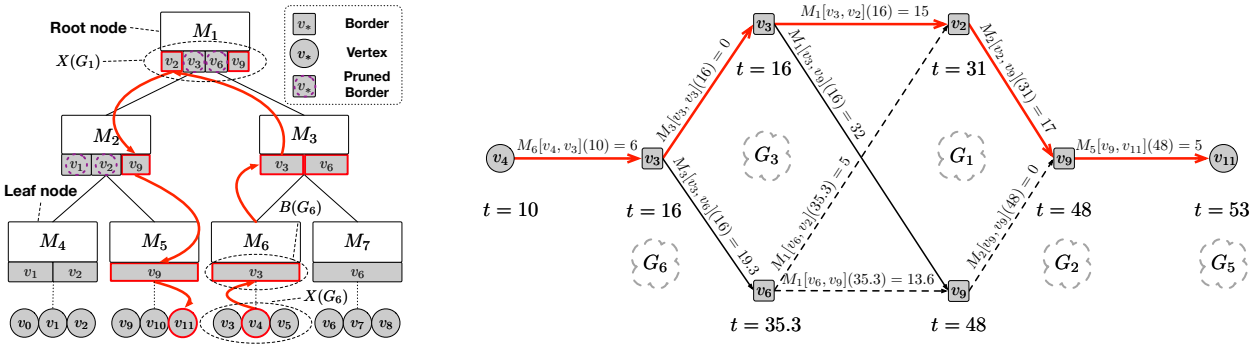
[Travel Time Initialization.] Initialize shortest travel time from v_s to pivot borders in $B(G_1) = X(G_1) \cap X(G_2)$ (line 7).

[Iterative Expansion.] Based on the shortest travel time to $B(G_1)$, it expands the shortest travel time to succedent tree node G_2 of G_1, i.e., vertices in $X(G_2)$, using dynamic programming (lines 8–12). To be more specific, let $X = X(G_1) \cap X(G_2)$, *i.e.,* the shared vertices set between M_1 and M_2 (line 10). Then, based on M_2 and shared vertices X, we push the shortest travel time from v_s to $v_b \in X(G_2) \setminus X(G_1)$ (line 11). Repeat the above process by assembling the shortest travel time to vertices in succedent tree-node matrices, until we get the shortest travel time to vertices in $X(G_k)$, where leaf node G_k contains v_e. Finally, the shortest travel time from v_s to v_e is calculated (lines 12) , along with the incomplete path, *i.e.,* the part of the optimal path that consists of v_s, v_e, and pivot borders in the optimal path.

Note that during expanding shortest travel time to vertices in succedent vertex set $X(G_j)$, we only consider vertices in $X(G_j) \cap X(G_{j+1})$, i.e., hop over to pivot borders shared by the succedent partition. For example, for leaf node $G_1 = \text{leaf}(v_s)$, we only consider $B(G_1) = X(G_1) \cap X(G_2)$. This optimization safely prunes unpromising borders which need extra hops before connecting to borders in $X(G_{j+1})$.

Time Complexity. (1) If v_s and v_e are within the same leaf node, it takes at most $O(\kappa_l \cdot \log \kappa_l \cdot \log \alpha(T))$ for local Dijkstra search. (2) Otherwise, a query scans at most $2 \cdot \log_{\kappa_f} |V|/\kappa_l + 1$ tree nodes (matrices) from leaf(v_s) to root and then to leaf(v_e). In the worst case without pruning, we expand all $O(\log_2 \kappa_f \cdot \sqrt{|V|/\kappa_f{}^{i-1}})$ borders in each passing matrix in the i-th level. The time cost is $O(2 \cdot \sum_{i=1}^{H} (\log_2 \kappa_f \cdot \sqrt{|V|/\kappa_f{}^{i-1}} \cdot \log_2 \alpha(T))^2) = O(\log_2^2 \kappa_f \cdot |V| \cdot \log_2^2 \alpha(T))$.

Example 13: [TDSP.] Consider a TDSP query $Q = \langle v_4, v_{11}, 10\rangle$ and Figure 4, where leaf(v_4) = G_6, leaf(v_{11}) = G_5, LCA(v_4, v_{11}) is G_1, and the corresponding tree-node path is G_6, G_3, G_1, G_2, G_5, as shown in Figure 8(a).



(a) Tree-node path (b) Assembly-based dynamic programming

Figure 8: A TDSP Example

Referring to Figure 8(b), we initialize shortest travel time to forwarding border vertices in $B(G_6)$, *i.e.,* $\tau^*_{v_4v_3}(10) = M_6[v_4, v_3](10) = 6$. Then we expand shortest travel time to $X(G_3)$, *i.e.,* v_3 and v_6. When we expand to $X(G_1)$ with four borders, where v_3 and v_6 in $X(G_1)$ are pruned since they are not in succeeding $X(G_2)$ (marked by dashed circles in Figure8(a)), *i.e.,* these two borders are not direct pivot vertices to the destination. Thus, we only expand to v_2 and v_9, where $\tau^*_{v_4v_2}(10) = \min\{\tau^*_{v_4v_3}(10) + M_1[v_3, v_2]\ (10 + \tau^*_{v_4v_3}(10)), \tau^*_{v_4v_6}(10) + M_1[v_6, v_2]\ (10 + \tau^*_{v_4v_6}(10))\} = 15$, where optimal and non-optimal relaxations are shown in solid lines and dashed lines in Figure 8(b), respectively. Similarly, $\tau^*_{v_4v_9}(10) = \min\{\tau^*_{v_4v_3}(10) + M_1[v_3, v_9](10 + \tau^*_{v_4v_3}(10)), \tau^*_{v_4v_6}(10) + M_1[v_6, v_9](10 + \tau^*_{v_4v_6}(10))\} = 38$.

Repeating the expanding process, when expanding to G_2, we prune borders v_1 and v_2 because they are not in succeeding leaf node G_5. Finally, we get $\tau^*_{v_4v_{11}}(10) = 43$. Hence, we have $\texttt{TDSP}(v_4, v_{11}, 10)$=43, and reversely retrieving along solid lines we get the incomplete border path $\langle v_4, v_3, v_2, v_9, v_{10}, v_{11}\rangle$ (red bold lines in Figure8(b)). □

4.2 TIP Query

To answer a TIP query, a straightforward way is to discrete the time interval into multiple points and execute the TDSP algorithm for every time point $t \in I$. However, this method is rather expensive since there are too many time points (even uncountable). To address this issue, we generalize Algorithm 4 in the TDSP problem to handle TIP queries more efficiently. Before introducing the algorithm for TIP, we first introduce PLF based segmentation. For the other two operations Com() and Min(), please refer to Section 3.2.

Seg(f, I). To calculate TIP for given time interval $I \subseteq T$, we need to intersect travel time functions on I, which is called *segmentation*. Given a travel time function f and a certain time interval $I \subseteq T$, we cut out the sub function during time domain I, *i.e.,* corresponding interpolation points. The segmentation operation costs $O(\alpha(T))$.

TIP Algorithm. The algorithm for processing a TIP query is shown in Algorithm 5, which returns the optimal departure time t^*, its corresponding shortest travel time, and the incomplete optimal path.

Similarly to TDSP, we also consider two cases.

(1) v_s and v_e are in the same leaf node. In this case, we can answer this query by directly invoking the Seg() function (lines 2–3). Note that a non-border pair of (v_s, v_e) is handled by local TD-Bellman-Ford based method (TD-Dijkstra based approaches that incur heavy time-expanded queue operations are not efficient on time-interval search).

Algorithm 5: TIP

Input: $Q = (v_s, v_e, I = [t_s, t_e])$, and a TD-G-tree $\mathbf{T}$
Output: Best departure time, corresponding shortest travel time, and its *incomplete* optimal path

1 Locate leaf(v_s) and leaf(v_e) by a hash table;
2 **if** leaf(v_s) = leaf$(v_e) = G_L$ **then**
3 $\quad \tau^*_{v_sv_e}(t) = \text{Seg}(M_L[v_s, v_e], I)$ where $t \in I$;
4 **else**
5 $\quad G_1$=leaf(v_s), G_k=leaf(v_e), $G_A = \text{LCA}(G_1, G_k)$;
6 $\quad$ Find the tree path $G_1 \cdots G_A \cdots G_k$ in $\mathbf{T}$;
7 $\quad \{\tau^*_{v_sv_b} = \text{Seg}(M_1[v_s, v_b], t \in I), v_b \in B(G_1)\}$;
8 $\quad$ **foreach** $j = 1$ *to* $k - 1$ **do**
9 $\quad\quad$ **foreach** $v_b \in X(G_{j+1}) \setminus X(G_j)$ **do**
10 $\quad\quad\quad X = X(G_j) \cap X(G_{j+1})$;
11 $\quad\quad\quad$ **foreach** $v_x \in X$ **do**
12 $\quad\quad\quad\quad \tau'_{v_sv_b} = \tau^*_{v_sv_x} + \text{Com}(M_{j+1}[v_x, v_b], \tau^*_{v_sv_x})$;
13 $\quad\quad\quad\quad \tau_{v_sv_b} = \text{Min}(\tau_{v_sv_b}, \tau'_{v_sv_b})$;
14 $\quad$ **foreach** $v_b \in B(G_k)$ **do**
15 $\quad\quad \tau'_{v_sv_e} = \tau^*_{v_sv_b} + \text{Com}(M_k[v_b, v_e], \tau^*_{v_sv_b})$;
16 $\quad\quad \tau_{v_sv_e} = \text{Min}(\tau_{v_sv_e}, \tau'_{v_sv_e})$;
17 $t^* = \arg\min_{t \in I}\{\tau^*_{v_sv_e}(t)\}$;

(2) v_s and v_e are in different leaf nodes. We assemble travel time function $\tau^*_{v_sv_e}$ on I by utilizing travel time matrices along tree-node path from G_1 to G_k (lines 5–13).

We first initialize shortest travel time functions to vertices in $B(G_1) = X(G_1) \cap X(G_2)$ (line 7). Then, based on the shared vertex set (line 10), we assemble STTFs to vertices in succeeding $X(G_2)$, and generalize shortest travel time function $\tau^*_{v_sv_b}$ based on all viable $M_2[v_x, v_b]$ for v_x in the intersection of matrix vertices (line 11–13).

Repeat assembling process to succeeding tree nodes until we get shortest travel time functions to G_k (lines 8–13). Finally we get function $\tau^*_{v_sv_e}$ by minimization with entries of M_k to v_e(lines 14-16). Retrieving t^* from $\tau^*_{v_sv_e}(t), t \in I$ with shortest travel time during I, we get optimal departure time t^* (line 17).

Time Complexity. Suppose we need at most $\alpha(I)$ linear pieces to represent related PLFs on interval I. Finding optimal departure time costs $O(\alpha(I))$. Thus, based on time complexity of TDSP, we get time cost $O(\log_2^2 \kappa_f \cdot |V| \cdot \alpha(I)^2)$.

Example 14: [TIP.] Consider a TIP query $Q = (v_4, v_{11}, I = [0, 60])$. Based on the assembly-based dynamic programming process designed for travel time functions, we first get $\tau^*_{v_4v_{11}}(t), t \in [0, 24.9]$ (shown in Equation 1), where departure time interval $(24.9, 60] \subset I$ is eliminated, because its corresponding arrival time interval $(60, *)$ exceeds the time domain $[0, 60]$ being considered. The corresponding optimal path for every sub time interval is in Equation 2, whose re-

covery details will be discussed in Section 4.3. As we can see, adjacent sub time intervals $[11.4, 24)$ and $[24, 24.9]$ from $\tau^*_{v_4v_{11}}(t)$ are merged since they correspond to the same optimal path $\langle v_4, v_3, v_6, v_9, v_{10}, v_{11}\rangle$. It is not hard to see the optimal departure time $t^* = 24$, where the corresponding shortest travel time is $35mins$.

$$\tau^*_{v_4v_{11}}(t) = \begin{cases} 0.8t + 39.8 & \text{if } t \in [0, 4) \\ 43 & \text{if } t \in [4, 11.4) \\ -0.63t + 43 & \text{if } t \in [11.4, 24) \\ 0.1t + 35 & \text{if } t \in [24, 24.9] \end{cases} \quad (1)$$

$$\rho^*_{v_4v_{11}}(t) = \begin{cases} \langle v_4, v_3, v_2, v_1, v_9, v_{10}, v_{11}\rangle & \text{if } t \in [0, 4) \\ \langle v_4, v_3, v_2, v_0, v_1, v_9, v_{10}, v_{11}\rangle & \text{if } t \in [4, 11.4) \\ \langle v_4, v_3, v_6, v_9, v_{10}, v_{11}\rangle & \text{if } t \in [11.4, 24.9] \end{cases} \quad (2)$$

□

4.3 Path Recovery

As discussed earlier in this section, algorithms for TDSP and TIP only return an incomplete optimal path. This is because that when computing the shortest path in those algorithms, we "hop over" detailed sub path between adjacent pivot borders. Next we will discuss how to recover the complete shortest path from the incomplete one.

4.3.1 Matrix Entry Path Recovery

To support path recovery, during calculating travel time matrix entries with TD-Floyd (see Algorithm 1), we record the intermediate vertex information. Specifically, when the algorithm relaxes a linear function piece f of travel-time function $M_i[v_s, v_e]$ with $M_i[v_s, v_k]$ and $M_i[v_k, v_e]$, if f is updated, it records the intermediate vertex as v_k for f. We use $\texttt{Inter}(M_i[v_s, v_e]), t)$ to denote the intermediate vertex at t.

Path-Recovery Algorithm. For $M_i[v_s, v_e](t), t \in T$, by retrieving the intermediate vertex information recursively we get the detailed optimal path from v_s to v_e departing at t. Algorithm 6 shows the pseudo-code. Based on the intermediate vertex conditions, there are following cases.

1. *v_s and v_e are non-border vertices (lines 1–2).* v_s, v_e are local vertices in G_i, where we know $\tau^*_{v_s,v_e}(t)$ during "hopping over" its sub-path. We adopt travel time bounded TD-Dijkstra to retrieve the detailed local path, e.g., from v_{10} to v_{11}.
2. *Directly connected (lines 5-6).* The optimal path from v_s to v_e is connected directly by an edge, *e.g.,* $M_4[v_2, v_1]$ is connected by $e_{v_2v_1}$ during $[0, 25]$ (see Figure 3(b)).
3. *Connected via $v_x \in X(G_i)$ (lines 7-8).* Note that intermediate vertex v_x is also maintained associated to M_i. We recover connection information for $M_i[v_s, v_x](t)$ and $M_i[v_x, v_e](t + M[v_s, v_x](t))$ recursively, *e.g.,* $M_4[v_2, v_1](t)$ is connected via v_0 during $[25, 60]$ (see Figure 3(b)).
4. *Connected via an inside vertex in $V_i \setminus X(G_i)$ (lines 9-10).* That is, intermediate vertex v_x is maintained by a child-node matrix M_c, *i.e.,* we synchronize $M_i[v_s, v_e] = M_c[v_s, v_e]$ in Approximate: Bottom-up Local-optimal, *e.g.,* $M_2[v_2, v_1] = M_4[v_2, v_1]$, where intermediate vertex is $v_0 \notin X(G_2)$ during $[25, 60]$.
5. *Connected via an outside vertex in $V \setminus V_i$ (lines 11-12).* Intermediate vertex v_x is maintained by parent-node matrix M_f, *i.e.,* we update $M_i[v_s, v_e] = M_f[v_s, v_e]$ during Precise: Top-Down Global-optimal, *e.g.,* $M_3[v_3, v_6] = M_1[v_3, v_6]$, with inter-vertex $v_2 \notin V_3$ during $[0, 15]$.

Algorithm 6: PREC(v_s, v_e, t, G_i)

Input: $v_s, v_e \in X(G_i)$, dpt. time t, and tree node G_i
Output: A full path from v_s to v_e

```
1  if leaf(v_s) = G_i and v_s, v_e ∉ B(G_i) then
2      return TD-Dijkstra(v_s, v_e, t); // τ*_{v_s,v_e}(t) local bounded
3  marker = Inter (M_i[v_s v_e], t);
4  if marker = v_x ∈ X(G_i) then
5      if v_x = v_s then
6          return ∅;
7      t_x = t + M_i[v_s, v_x](t);
8      return <PREC(v_s,v_x,t, G_i), v_x, PREC(v_x,v_e, t_x, G_i)>;
9  else if marker = G_f then // parent node G_f
10     return PREC(v_s, v_e, t, G_f);
11 else if marker = G_c then // child node G_c
12     return PREC(v_s, v_e, t, G_c);
```

Note that different departure time points during the same linear function segment has the same intermediate vertex. Hence, we only need to maintain one intermediate vertex information per linear segment for all matrix entries.

For TDSP query $Q(v_s, v_e, t)$ (Section 4.1), we need to recover the optimal path corresponding to departure time t. For TIP query $Q(v_s, v_e, I)$, what we concern is the optimal departure time t^* within I and its corresponding optimal path. Hence, we only discuss the path-recovery algorithm for TDSP queries, which can also support TIP queries.

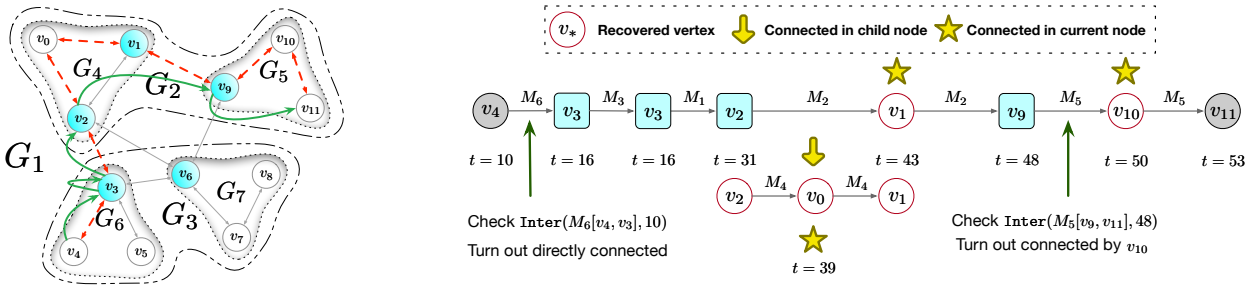
4.3.2 Chronological Path-Recovery Process

For a TDSP query $Q(v_s, v_e, t)$, based on the shortest travel time, we retrieve the incomplete optimal path (called border path) $\langle v_s, b_1, b_2, b_3, \cdots, b_m, v_e\rangle$ corresponds to the tree-node path $\mathsf{leaf}(v_s) = G_1, \cdots, \mathsf{LCA}, \cdots, G_m = \mathsf{leaf}(v_e)$ (see Example 13). Based on the border path and corresponding shortest travel time passing these borders, we recover the detailed inter-vertices between each adjacent vertex pair in the border path chronologically as shown in Algorithm 6. More concretely, for each adjacent pair (b_{i-1}, b_i) passing from G_{i-1} to G_i, we use t_c to represent the current time arriving at b_{i-1}. Then we recover the detailed path of $M_i[b_{i-1}, b_i]$ at t through Algorithm 6. As mentioned in Section 3.1, non-border vertex pairs (v_i, v_j) in leaf-node matrices are not maintained for space consideration. Alternatively, we compute the optimal sub-path via (v_i, v_j) by local TD-Dijkstra search.

Example 15: [Path Recovery.] Consider a TDSP query discussed in Example 13, $Q(v_4, v_{11}, 10)$. Reversely retrieving the shortest travel time along the way based on shortest travel time $43mins$ (reversely along bold solid lines in Figure 8(b)), we get the optimal border path $\langle v_4, v_3, v_2, v_9, v_{11}\rangle$.

Referring to the border path shown in Figure 9(a) with bold solid lines, we see the detailed path between each two adjacent elements in the border path are not recovered, *e.g.,* the subpath between v_2, v_9, and the sub path between v_9, v_{11} (dashed lines in Figure 9(a)).

Then, referring to detailed path recovery process in Figure 9(b), let's focus on adjacent pair (v_2, v_9) on tree node G_2 with departure time $t = 31$. Checking $\texttt{Inter}(M_2[v_2, v_9], t = 31)$ (line 3), v_2 and v_9 are connected via v_1 in current node (line 4, circled vertex with star superscript in Figure 9(b)). We get arrival time $t_x = t + M_2[v_2, v_1](31) = 43$ at v_1 (line 7). Recursively, we process (v_2, v_1) and (v_1, v_9) (line 8), where (v_2, v_1) are connected in G_2's child node G_4 (line 11, and the down arrow from M_2 to M_4 in Figure 9(b)). Similarly, to



(a) Incomplete optimal path (border path) (b) Vertex-by-vertex path recovery

Figure 9: Path recovery of $Q(v_4, v_{11}, 10)$

Table 3: Datasets

Dataset	#Vertices	#Edges
CAL (California)	21,048	43,386
SF (San Francisco)	174,956	446,002
COL (Colorado)	435,666	1,057,066
FLA (Florida)	1,070,376	2,712,798
W (Western USA)	6,262,104	15,248,146

process non-border pair (v_{10}, v_{11}), we first get shortest travel time bound $\tau^*_{v_{10},v_{11}}(50) = \tau^*_{v_4,v_{11}}(10) - \tau^*_{v_4,v_{10}}(10) = 3$. With TD-Dijkstra local search bounded by $3mins$ we get v_{10} and v_{11} are connected directly (line 2). □

Space and Time Complexity. Maintaining Inter for a PLF costs $O(\alpha(T))$. As there are $O(\log_{\kappa_f} \frac{|V|}{\kappa_l} \cdot \log_2^2 \kappa_f \cdot |V|)$ matrix entires (Section 3.2), the space complexity is $O(\log_{\kappa_f} \frac{|V|}{\kappa_l} \cdot \log_2^2 \kappa_f \cdot |V| \cdot \alpha(T))$. Consider an adjacent pair (v_s, v_e) on tree node G_i with departure time t_c at v_s, checking $M_i[v_s, v_e]$ for t_c costs $O(\log_2 \alpha(T))$. Suppose there are N_p inter-vertices to be recovered, travel time bounded TD-Dijkstra local sub-path search costs $O(C_{\kappa_l})$, a κ_l related constant, the overall time cost is $O(N_p \cdot C_{\kappa_l} \cdot \log_2 \alpha(T))$.

5. EXPERIMENTS

Datasets: We used five real-world road-network datasets [2,3], which were directed graphs. The numbers of vertices in the datasets were varied from 20,000 to 6,262,104. All the time-dependent weight functions (PLFs) were generated based on real traffic patterns that were gotten from real taxi trajectories [18]. We set the time domain as a whole day, i.e., 86400 seconds. The average number of interpolation points in the time-dependent weight functions for each edge was 3. The statistics of these datasets were shown in Table 3.

Setting: For TD-G-tree, we set fanout κ_f to 4 and the maximal number κ_l of vertices in a leaf node is considered in Section 5.1. Original graph inputs and the TD-G-tree index are memory resident throughout the following experiments.

TDSP. (1) To evaluate the efficiency of processing a TDSP query, we randomly chose 1,000 pairs of vertices for each dataset. For each pair of vertices, we randomly chose 10 departure time, and thus we had 10,000 TDSP queries for each dataset. (2) To evaluate the scalability of TDSP, we also varied the number of hops from the source to the destination. (3) To further analyze query efficiency, we evaluated the number of visited (labeled) vertices during TDSP query processing, we recorded the number of visited vertices for queries in (1) and (2).

[2]https://www.cs.utah.edu/∼lifeifei/SpatialDataset.htm
[3]http://www.dis.uniroma1.it/challenge9/download.shtml

TIP. (1) To evaluate the performance of a TIP query, we randomly chose 1,000 random pairs of vertices for each dataset and 10 time intervals $I = [t_s, t_e] \subseteq T$, where $|t_e - t_s|$ was uniformly selected from $0.1|T|$, $0.2|T|$, $\cdots$, $|T|$ ($|T|$= 86400 seconds). (2) To evaluate the influence of time-interval length on TIP queries, we varied time interval size from 10 to 10,000 seconds, and for each time interval we generated 1,000 queries randomly.

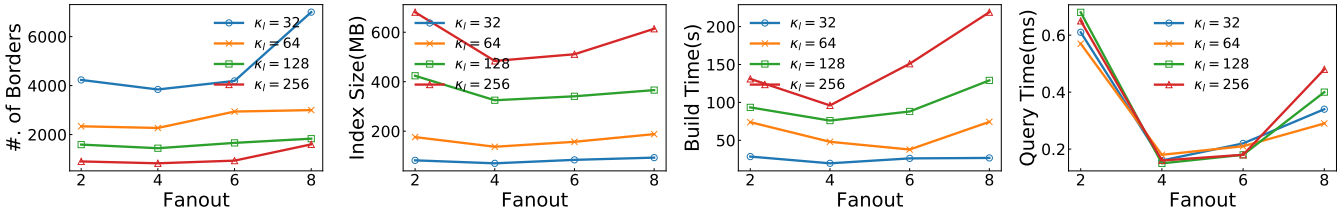
Update. To evaluate the update efficiency of TD-G-tree, we randomly chose 0.1%, 1%, 10% of edges, and we randomly decreased the speed of high-speed edges, and increased the speed of low-speed edges.

Comparisons. For TDSP queries, we compared our TD-G-tree with TDALT [19] (A^* with landmarks and triangle inequalities), TCH [2] (time-dependent version of contraction hierarchy which we implemented based on TCH [13] and CH [2]) and TD-Dijkstra, which extended the Dijkstra algorithm to support time-dependent road networks. For TIP queries, we compared with TCH and TD2S [10], which was a Dijkstra-based method. TD2S avoided redundant vertices relaxations for every time point, and guaranteed that the shortest path of a vertex can be found if other vertices cannot be arrived before the current vertex.

All the algorithms were implemented in C++. All experiments were conducted on a Linux computer with Intel 2.20GHz CPU and 128GB memory.

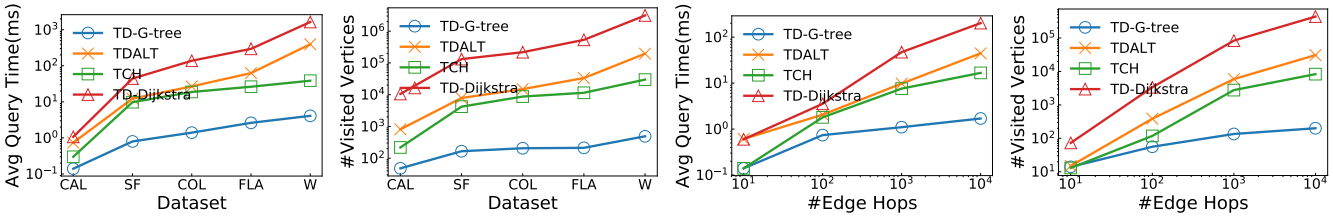
5.1 Evaluation on Parameters: κ_f and κ_l

We evaluated the impact of κ_f (partition fanout) and κ_l (the maximum number of vertices in each leaf node) of TD-G-tree by investigating the number of borders generated by the hierarchical partitioning, the index size, the index building time and the TDSP query time. We varied κ_f in $\{2, 4, 6, 8\}$ and κ_l in $\{32, 64, 128, 256\}$. Figure 10 showed the results. We made three observations. (1) Our method achieved better results with medium $\kappa_f = 4$ or 6. With the increasing of partition fanout κ_f, the border number, index size, build time and query time decreased first and then increased. It is because increasing fanout will increase tree nodes number per level and reduce tree height, but larger fanout will generate more borders which lead to more matrix entries. (2) With the increase of κ_l, the number of borders decreased as hierarchical partitioning stopped earlier and there are less partitions. But index size and build time increased for CAL, since bigger κ_l leads to more borders per leaf node G_L, *i.e.,* more square-level matrix entries in M_L, which lead to longer building time. (3) Our method achieved the best results when $\kappa_l = 64$ for optimizing query time, because larger κ_l lead to lower tree height with lesser searching tree nodes, but it caused longer local Dijkstra search time on query pairs within the same leaf node. To balance the



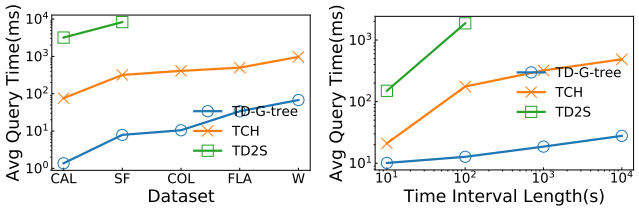
(a) Number of borders (b) Index size(MB) (c) Index build time(s) (d) Query time(ms)

Figure 10: Evaluation on parameters: f and κ_l (CAL dataset)



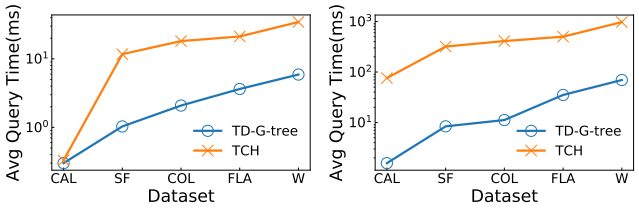
(a) Random query (b) Random query vis# (c) Vary edge hops (FLA) (d) Edge hops (FLA) vis#

Figure 11: Comparison on TDSP query processing



(a) Random query (b) Vary time interval(FLA)

Figure 12: Comparison on TIP query processing



(a) Random TDSP-PREC (b) Random TIP-PREC

Figure 13: TDSP and TIP with path recovery

number of tree nodes per level and corresponding matrix size during partitioning, we set $\kappa_f = 4$. To balance global tree-node level and local Dijkstra searching, we set κ_l to 64(CAL), 64(SF), 128(COL), 256(FLA) and 256(W).

5.2 Evaluation on Query Efficiency

Evaluation on TDSP Search: We evaluated the TDSP query efficiency and compared TD-G-tree with TDALT, TCH and basic TD-Dijkstra. Figure 11 showed the results.

We first evaluated the performance by randomly choosing a pair of vertices from each dataset and uniformly chose the departure time in $[0, 86400]$. Figure 11(a) showed the performance. We could see that our method outperformed existing methods by 1-2 orders of magnitude. This was because Dijkstra was rather expensive to find the shortest paths by expanding the vertices in the graph, and without indexed shortest travel time functions between special border vertices. Although TDALT eliminated many vertices through lightweight indexed heuristic landmarks, it still needs to traverse along the shortest path, which took quite a long time. TCH also took much longer time than TD-G-tree, because TD-G-tree used the effective index to search the shortest path that greatly reduced visited vertices. Besides, TDSP query time grew very slowly as the dataset became larger, e.g., the query time increased from 2.6ms (FLA with 1.1 million vertices) to 4.1ms (W with 6.2 million vertices). Second, we varied the number of hops from the sources to the destinations on FLA. Figure 11(c) showed the results. By varying the number of hops on different datasets, TDALT, TCH and TD-Dijkstra had worse performance while TD-G-tree showed more stable performance. This was because compared techniques required to traverse many more vertices while TD-G-tree used the index structure to skip a large number of unnecessary vertices. Third, we recorded visited vertices for random queries and hops by varying queries discussed above. Figures 11(b) and 11(d) showed the numbers. Compared with TDALT, TCH and TD-Dijkstra which need to search many vertices, TD-G-tree based shortest travel time querying only need to settle a few pivot borders, e.g., only several hundreds of vertices on W.

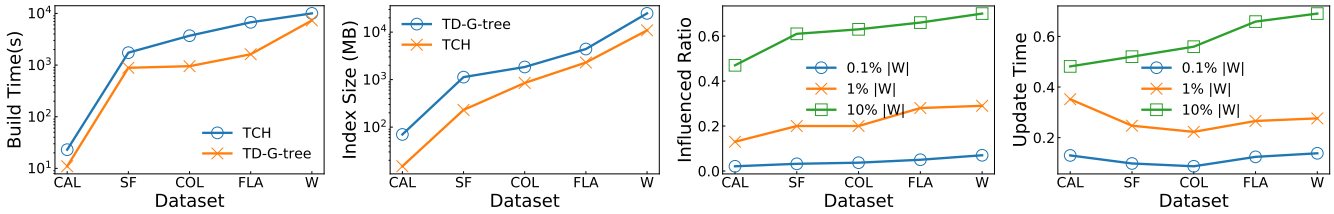
Evaluation on TIP Search: We evaluated the TIP query efficiency and compared TD-G-tree with TCH [2] and TD2S [10].

First, we tested the performance by randomly choosing query vertex pairs and randomly choosing time intervals in $[0, 86400]$. Figure 12(a) showed the results. TD-G-tree was 1–2 orders of magnitude faster than TCH, and three orders of magnitude faster than TD2S. This was because $Dijkstra$-like search for TD2S and label correcting approach for TCH were rather expensive for large interval based TIP queries. Moreover, with the increase of the dataset, the gap between them became larger (and TD2S could not support large datasets). Second, we fixed the query vertex pairs and varied the TIP query time interval length on dataset FLA. Figures 12(b) showed the results. We could find that TIP outperformed the baselines by more than one order of magnitude. Moreover, with the increase of the query time interval length, TD2S took much more time, because it required to search longer paths while TD-G-tree was more stable, as TD-G-tree has indexed the whole time domain and may segment the needed intervals.

Evaluation on Path Recovery. We evaluated our path recovery scheme on random queries of TDSP and TIP discussed in Section 5.2 and compared with TCH (as shown in Figure 13). Our method still achieved higher performance when combined with path recovery, because our method can efficiently retrieve the shortest paths based on pivot borders.

5.3 TD-G-Tree Construction and Update

Index Building: We evaluated the time and space overhead for indexing, and we compared TD-G-tree with TCH.



(a) Build time(s) (b) Index size(MB) (c) Update: # tree nodes (d) Update: time cost(s)

Figure 14: Index building and update

Figure 14(a)-14(b) illustrated the index building time and the index size. Note that, the index of `TD-G-tree` included tree structure, travel time matrices and their intermediate vertices, hash table that mapped vertices to the corresponding leaf nodes. `TD-G-tree` achieved similar building time with `TCH`, where `TD-G-tree` generated shortest travel time matrices, and `TCH` contracted the vertices according to time varying importance. `TD-G-tree` had larger index size than `TCH`, as `TD-G-tree` indexed more matrices for borders. Fortunately, the index sizes are acceptable for all road networks.

Index Update: Existing work cannot support the index update for `TDRNs`. Here we evaluated `TD-G-tree` update efficiency. We varied the update ratio and the result was shown in Figure 14(c)-14(d). As we can see from the results, varying part of the edge weights being updated, i.e., from 0.1%, 1% to 10% of total number of edges, only no more than 20% of tree nodes (travel time matrices) were influenced by the update of $0.1-1\%$ of edges, and the update time overhead was rather small, i.e., no more than 15% for the building time on 0.1% of edges' update. Besides, with the increase of the update ratio, the update time increased heavily. But in real traffic, there would not be many updated edges and our method was acceptable in real scenarios.

6. RELATED WORK

Many efficient shortest path algorithms have been studied for static road networks [1, 9, 13, 38, 36, 31, 37, 32, 20, 33]. We now consider studies on shortest path problems in time-dependent road networks, *i.e.,* `TDSP` problems (see [26] for an introduction, and [14, 24] for an overview).

TDSP. `TDSP` was first proposed in [4] by Cooke and Helsey with a recursion formula. Dreyfus[11] generalized Dijkstra to time-dependent graphs with implicit first-in-first-out (`FIFO`) property (*i.e.,* not-overtaking) assumption. Kaufman and Smith [23] proved `TDSP` is polynomially solvable in `FIFO` networks. Non-`FIFO` networks are proved to be NP-hard by Orda and Rom [30]. Fortunately, the transportation road networks satisfy the `FIFO` property, which guarantees the problems can be solved in a polynomial time. Thus, we mainly focus on graphs with the `FIFO` property. Note that our techniques can be extended to solve non-`FIFO` time-dependent graphs if waiting at vertices is permitted (*i.e.,* polynomial solvable).

Generalized Classic Techniques. Orda and Rom [30] extended label-correcting method to time-dependent scenarios, with a time complexity of $O(\mathcal{F}_{\max}|V||E|)$, where $\mathcal{F}_{\max}$ is the maximum output size for all possible destination nodes. Ding et al. [10] and Dehne et al. [5] presented two label-setting Dijkstra like algorithms, which has time overhead $O((|V|\log|V|+|E|)\alpha(T))$, where $\alpha(T)$ is linear pieces needed to represent `PLFs` on time domain T.

A^*-based Algorithms. The work [21] extended A^* algorithm to time-dependent road networks, where expanding all possible paths may incur exponential running time in the worst case. A^* with landmarks (`ALT` [15]) and bidirectional search scheme in A^* [16] were combined to handle time-dependent scenarios in [8, 29]. Two-level hierarchical approach was combined to bidirectional `ALT` to obtain time-dependent core `ALT` (`TDCALT`) in [7].

Hierarchical Techniques. `SHARC` [3] and `CH` [13] were two hierarchical speed-up techniques in static road networks, which were augmented to time-dependent road networks as `TD-SHARC` [6] and `TCH` [2] respectively. The main idea behind `TCH` and `TD-SHARC` is to remove unimportant vertices and generate shortcuts between remaining important vertices.

Variants of Shortest Path Querying on `TDRNs`. There are also other variant works, either on `TDRN` modeling or optimizing targets. [12] developed a `TDRN` dataset using sparse trajectories, which also showed `TDSP` querying is more desirable than static shortest path querying. [19, 28] turned deterministic `TDRNs` models into probabilistic models with edge travel time assigned by probability distribution functions, based on which they proposed and handled probability based queries. [17] considered `TDSP` problems under constraints such as greenhouse gas emissions and congestion costs by using `TD-Dijkstra` to calculate shortest paths. R3 [34] discretized continuous `TDRNs` into static snapshots under different traffic and recommend routes by integrating these static road networks together. [35] studied time-dependent route planning problems on public transportation networks, *e.g.,* routing among bus stops based on accessible road segments. [25, 27] studied route scheduling that minimize on-road time on `TDRNs` by allowing parking on intermediate vertices.

7. CONCLUSION

We have studied the problem of answering shortest path queries on `TDRNs`. We proposed a balanced tree-structured index, called `TD-G-tree`, to support both shortest path queries and time interval planing queries. We used hierarchical graph partitioning to split the road network into hierarchical partitions and constructed a balanced tree index. We further proposed effective algorithms to support the two types of queries. Experiments showed that our method significantly outperformed existing studies, *e.g.,* about 8 times faster on `TDSP` and 10 times faster on `TIP` queries. For future work, we aim to support more types of `TD-G-tree`-based queries, *e.g.,* k-nearest neighbors query (k-NN) for points of interests (POIs). We want to explore more precise methods for real-time traffic flow prediction, which helps `TD-G-tree` based systems to better handle traffic emergencies.

Acknowledgement. This paper was supported by 973 Program (2015CB358700), NSF of China (61632016, 61521002, 61661166012), Huawei, and TAL education.

8. REFERENCES

[1] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. In *Algorithm engineering*, pages 19–80. 2016.

[2] G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-dependent contraction hierarchies. In *ALENEX*, pages 97–105, 2009.

[3] R. Bauer and D. Delling. Sharc: Fast and robust unidirectional routing. *JEA*, 14:4, 2009.

[4] K. L. Cooke and E. Halsey. The shortest route through a network with time-dependent internodal transit times. *Journal of mathematical analysis and applications*, 14(3):493–498, 1966.

[5] F. Dehne, M. T. Omran, and J.-R. Sack. Shortest paths in time-dependent fifo networks. *Algorithmica*, 62(1-2):416–435, 2012.

[6] D. Delling. Time-dependent sharc-routing. In *ESA*, pages 332–343, 2008.

[7] D. Delling and G. Nannicini. Core routing on dynamic time-dependent road networks. *INFORMS JOC*, 24(2):187–201, 2012.

[8] U. Demiryurek, F. Banaei-Kashani, C. Shahabi, and A. Ranganathan. Online computation of fastest path in time-dependent spatial networks. In *SSTD*, pages 92–111. Springer, 2011.

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math*, 1(1):269–271, 1959.

[10] B. Ding, J. X. Yu, and L. Qin. Finding time-dependent shortest paths over large graphs. In *EDBT*, pages 205–216, 2008.

[11] S. E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations research*, 17(3):395–412, 1969.

[12] E. Eser, F. Kocayusufoğlu, B. Eravci, H. Ferhatosmanoğlu, and J. L. Larriba-Pey. Generating time-varying road network data using sparse trajectories. In *ICDMW*, pages 1118–1124. IEEE, 2016.

[13] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

[14] M. Gendreau, G. Ghiani, and E. Guerriero. Time-dependent routing problems: A review. *Computers & operations research*, 64:189–197, 2015.

[15] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, pages 156–165, 2005.

[16] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for a*: Shortest path algorithms with preprocessing. In *The Shortest Path Problem*, pages 93–140, 2006.

[17] H. Heni, L. C. Coelho, and J. Renaud. Determining time-dependent minimum cost paths under several objectives. *Computers & Operations Research*, 2019.

[18] H. Hu, G. Li, Z. Bao, Y. Cui, and J. Feng. Crowdsourcing-based real-time urban traffic speed estimation: From trends to speeds. In *ICDE*, pages 883–894. IEEE, 2016.

[19] M. Hua and J. Pei. Probabilistic path queries in road networks: traffic uncertainty aware path selection. In *EDBT*, pages 347–358. ACM, 2010.

[20] W. Huang and J. X. Yu. Investigating TSP heuristics for location-based services. *DSE*, 2(1):71–93, 2017.

[21] E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on A road network with speed patterns. In *ICDE*, page 10, 2006.

[22] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *SC*, pages 29–29. IEEE, 1995.

[23] D. E. Kaufman and R. L. Smith. Fastest paths in time-dependent networks for intelligent vehicle-highway systems application. *Journal of Intelligent Transportation Systems*, 1(1):1–11, 1993.

[24] K. Li and G. Li. Approximate query processing: What is new and where to go? *DSE*, 3(4):379–397, 2018.

[25] L. Li, W. Hua, X. Du, and X. Zhou. Minimal on-road time route scheduling on time-dependent graphs. *PVLDB*, 10(11):1274–1285, 2017.

[26] L. Li, J. Kim, J. Xu, and X. Zhou. Time-dependent route scheduling on road networks. *SIGSPATIAL Special*, 10(1):10–14, 2018.

[27] L. Li, K. Zheng, S. Wang, W. Hua, and X. Zhou. Go slow to go fast: Minimal on-road time route scheduling with parking facilities using historical trajectory. *The VLDB Journal*, 27(3):321–345, June 2018.

[28] X. Lian and L. Chen. Trip planner over probabilistic time-dependent road networks. *TKDE*, 2014.

[29] G. Nannicini, D. Delling, D. Schultes, and L. Liberti. Bidirectional a* search on time-dependent road networks. *Networks*, 59(2):240–251, 2012.

[30] A. Orda and R. Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *JACM*, 37(3):607–625, 1990.

[31] B. Shen, Y. Zhao, G. Li, W. Zheng, Y. Qin, B. Yuan, and Y. Rao. V-tree: Efficient knn search on moving objects with road-network constraints. In *ICDE*, pages 609–620, 2017.

[32] N. Ta, G. Li, T. Zhao, J. Feng, H. Ma, and Z. Gong. An efficient ride-sharing framework for maximizing shared route. *TKDE*, 30(2):219–233, 2018.

[33] E. Turunen. Using GUHA data mining method in analyzing road traffic accidents occurred in the years 2004-2008 in finland. *DSE*, 2(3):224–231, 2017.

[34] H. Wang, G. Li, H. Hu, S. Chen, B. Shen, H. Wu, W.-S. Li, and K.-L. Tan. R3: a real-time route recommendation system. *PVLDB*, 7(13):1549–1552, 2014.

[35] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou. Efficient route planning on public transportation networks: A labelling approach. In *SIGMOD*, pages 967–982. ACM, 2015.

[36] H. Yuan and G. Li. Distributed in-memory trajectory similarity search and join on road network. In *ICDE*, pages 1262–1273, 2019.

[37] R. Zhong, G. Li, K. Tan, and L. Zhou. G-tree: an efficient index for KNN search on road networks. In *CIKM*, pages 39–48, 2013.

[38] R. Zhong, G. Li, K. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. *TKDE*, 27(8):2175–2189, 2015.