Geoinformatica manuscript No.
(will be inserted by the editor)

Discovering Historic Traffic-tolerant Paths in Road Networks

**Pui Hang Li · Man Lung Yiu ·
Kyriakos Mouratidis**

Received: date / Accepted: date

Abstract Historic traffic information is valuable in transportation analysis and planning, e.g., evaluating the reliability of routes for representative source-destination pairs. Also, it can be utilized to provide efficient and effective route-search services. In view of these applications, we propose the k traffic-tolerant paths (TTP) problem on road networks, which takes a source-destination pair and historic traffic information as input, and returns k paths that minimize the aggregate (historic) travel time. Unlike the shortest path problem, the TTP problem has a combinatorial search space that renders the optimal solution expensive to find. First, we propose an exact algorithm with effective pruning rules to reduce the search time. Second, we develop an anytime heuristic algorithm that makes 'best-effort' to find a low-cost solution within a given time limit. Extensive experiments on real and synthetic traffic data demonstrate the effectiveness of TTP and the efficiency of our proposed algorithms.

Keywords Road Networks · Road Traffic

1 Introduction

Nowadays, historic traffic information is extensively collected from roadside sensors [2] and crowdsourcing techniques [4]. It is valuable in transportation reliability analysis [7, 20], e.g., evaluating the reliability of routes for representative source-destination pairs, and online route services [4]. In order to support these applications, we propose a novel problem called k traffic-tolerant paths (TTP). The idea of TTP is to extract a set of k paths $P^k_{s,d}$ from a road network such that it best approximates the shortest travel time for a given source-destination (SD) pair (v_s, v_d) at any time. Specifically, this problem requires a road network

Man Lung Yiu was supported by ICRG grant G-YN38 from the Hong Kong Polytechnic University. Kyriakos Mouratidis was supported by research grant 14-C220-SMU-004 from the Singapore Management University Office of Research under the Singapore Ministry of Education Academic Research Funding Tier 1 Grant.

P. H. Li (✉) · M. L. Yiu
Department of Computing, The Hong Kong Polytechnic University
E-mail: {csphli, csmlyiu}@comp.polyu.edu.hk

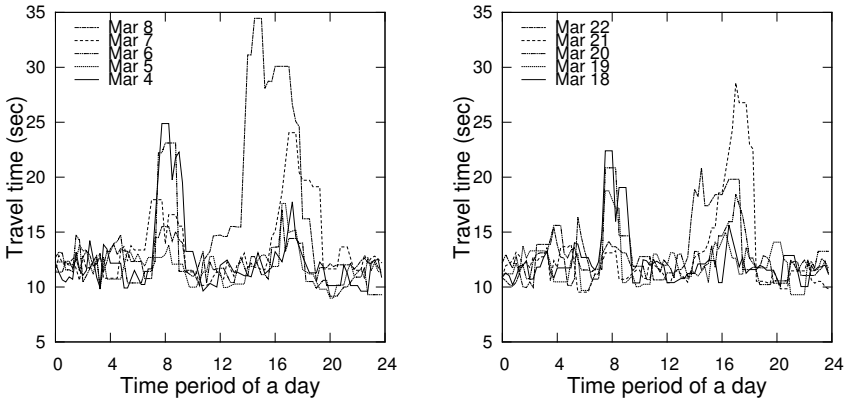
K. Mouratidis
School of Information Systems, Singapore Management University
E-mail: kyriakos@smu.edu.sg

$G(V, E, W_m)$ where W_m maps an edge to its travel time at m time instants. Given an integer k and a SD pair (v_s, v_d), the TTP problem is to find a set of k paths from v_s to v_d, denoted by $P^k_{s,d}$, that minimizes the following error:

$$\xi(P^k_{s,d}) = \frac{1}{m} \cdot \sum_{j=1}^{m} \left(\left(\min_{p \in P^k_{s,d}} \tau_j(p) \right) - \tau_j(sp_j) \right) \quad (1)$$

where $\tau_j(p)$ is the travel time of path p at time instant j and sp_j denotes the shortest path from v_s to v_d at time instant j.

The rationale behind Equation 1 can be explained with the aid of Figure 1. The figure shows the travel time along a UK highway road segment on weekdays of two weeks, demonstrating that major fluctuations occur in the morning and in the evening. Due to this similar and recurrent traffic pattern, fastest paths computed from historic traffic data are likely to be the fastest paths in the future and this can be further leveraged to answer and accelerate route queries. Inspired by this, we aim to extract a set of paths with the minimum travel time discrepancy from the historic fastest paths at any time. Equation 1 captures exactly that discrepancy. We present TTP applications in detail in Section 1.1.



(a) Mar 4-8, 2013 (b) Mar 18-22, 2013

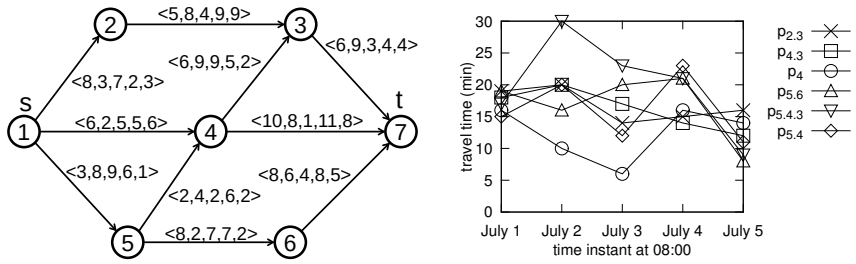
Fig. 1 Travel time of a road segment on A417 Road in UK [5]

Observe that the minimization of $\xi(P^k_{s,d})$ in Equation 1 is equivalent to the minimization of the following measure:

$$\Psi(P^k_{s,d}) = \sum_{j=1}^{m} \min_{p \in P^k_{s,d}} \tau_j(p) \quad (2)$$

because sp_j is independent of $P^k_{s,d}$ and the summation function is distributive.

We illustrate TTP by the sample road network in Figure 2(a). Each edge is labeled with $m = 5$ weights that represent the travel time of edges at 5 time instants (e.g., 8:00am on July 1 – 5 and i.e., $m = 5$). Suppose that the source-destination pair is $(v_s, v_d) = (v_1, v_7)$. There are 6 possible paths from v_s and v_d, and their travel time at different time instants are shown in Figure 2(b). For clarity, we indicate the intermediate nodes in a path in the subscript, e.g., the path $p_{5.4.3}$ passes through v_5, v_4, v_3. Assume that $k = 2$. The optimal path set is $P^k = \{p_4, p_{5.6}\}$ because it has the minimum $\Psi(P^k)$ value of 56.



(a) A road network (b) Travel time of paths

Fig. 2 Running example

1.1 Applications

TTP can be applied in *transportation planning analysis*. Transportation planners evaluate the reliability of transportation systems by analyzing the reliability of routes for representative SD pairs, which are chosen by their expertise. For example, representative source-destination pairs could be: city center to airport, port to the industrial area, etc. Their current practice [7,20] is to select only one route per SD pair and calculate the travel time reliability of each route. Our proposed TTP can provide k paths instead of a single path per SD pair to transport planners for reliability analysis. Since the k paths minimize historic aggregate travel time, they can be regarded as the alternatives from which planners can obtain a more comprehensive insight of the reliability of transportation systems. For example, four traffic-tolerant paths for a SD pair (shown in Figure 3) can be adopted by transport planners for reliability analysis.

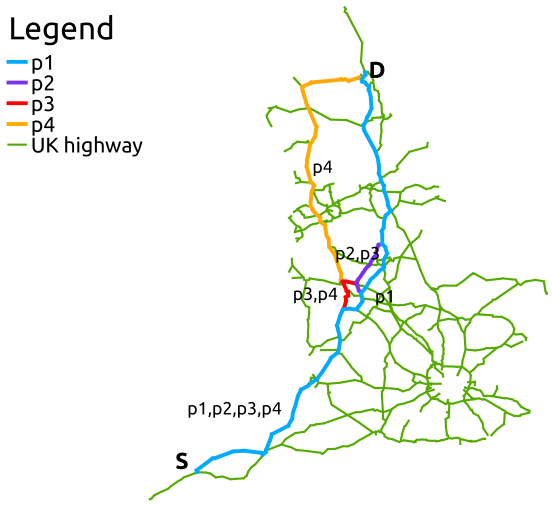


Fig. 3 Traffic-tolerant paths in UK highway network, $k = 4$

Besides, with the advancement and wide deployment of mobile devices and car navigation systems, online route services have access to real-time traffic information[1] and provide users with shortest path(s) according to up-to-date traffic. In fact, the majority of queries are *recurrent queries* issued by users daily, e.g., finding the fastest route from home to office at 8:00am every day. Such regular patterns appear in human trajectories, as revealed in current scientific studies [14, 25].

Although an online route service may apply shortest path indices [8, 9, 12, 15, 22–24, 27] to answer queries efficiently, such indices incur substantial maintenance costs[2] to deal with frequent traffic updates (e.g., every 30 seconds [2]), making it hard to answer shortest path queries with the latest traffic.

A promising method [21] is to pre-compute candidate paths (a solution pool) for users' recurrent queries (in an offline phase) and then update their travel time by live traffic information (during online operations). The candidate path with the shortest travel time is used to answer queries. This approach eliminates the index maintenance cost and bounds the online computation cost by the number of candidate paths. It scales better than existing methods based on shortest path indices.

In the above candidate path approach, it is desirable to find a set of k candidate paths per query (v_s, v_d) such that *at least one* of such paths is fast for (v_s, v_d) at any time. This problem is challenging due to ever-changing traffic conditions. Although traffic conditions change continuously, they exhibit some patterns which can be exploited for candidate path selection. For example, Figure 1 shows the travel time along a road segment on weekdays of two weeks. There are two obvious spikes during 8:00am – 9am and 6:00pm – 7:00pm in these two weeks.

With traffic time patterns of road segments, by minimizing the travel time error $\Psi(P^k_{s,d})$, we can obtain a set of traffic-tolerant paths (in Figure 3) and it is most likely that at least one of them equals to or is comparably fast to the actual fastest path at current time. Although Malviya et al. [21] have proposed some heuristics for finding these candidate paths, they do not necessarily minimize the historic travel time error between those heuristic paths and the fastest path in the road network.

1.2 Challenges and Contributions

Unlike the shortest path problem, the TTP problem has a combinatorial search space that renders the optimal solution expensive to compute; it needs to find the k paths that minimize the aggregate travel time in history among all possible paths for a SD pair. First, we propose an exact algorithm with effective pruning rules to reduce the search time. Second, we develop an anytime heuristic algorithm that makes 'best-effort' to find a low-cost solution within a given time limit. This paper is a substantial extension of our previous work [19]. In this paper, we elaborate our solutions in detail with examples, and give a proof on the hardness of TTP. In summary, our contributions are as follows.

- We propose a novel problem called k traffic-tolerant paths query (TTP), which finds application in transportation planning and online route services.
- We prove that the TTP problem is NP-hard.

[1] Collected from roadside sensors [2], crowdsourcing [4], or traffic information providers [6].

[2] The state-of-the-art shortest path index, AH [27], takes hundreds of seconds for index pre-computation on a road network with a million nodes.

- We present an exact algorithm with effective pruning rules that computes the optimal solution for TTP.
- We devise two heuristic algorithms for TTP. One of them offers an anytime feature which returns a reasonably good solution within a given time budget.

The rest of this paper is organized as follows. We first discuss the related work in Section 2. Subsequently, we formally define TTP in Section 3. We prove that the TTP problem is NP-hard in Section 4. Then, we present an exact algorithm in Section 5 and two heuristics in Section 6. Next, we elaborate on how to utilize TTP in applications in Section 7. We evaluate our TTP algorithms in Section 8 and finally conclude our work in Section 9.

2 Related Work

As we mentioned in the introduction, Malviya et al. [21] propose some heuristics to generate candidate paths for online route services with bounded cost. The heuristics do not necessarily minimize the travel time error between those candidate paths and the fastest paths in the historic data. In contrast, our TTP problem aims to minimize the historic travel time error in order to produce a set of candidate paths robust to the real-time traffic. Their work cannot be used for transportation reliability analysis as the heuristics may return some convoluted paths, which are not considered by transport planners.

Historic traffic data have already been used for route planning. Kanoulas et al. [17] and Demiryurek et al. [11] associate the traveling times of road segments with time-varying functions which are established based on historic data. The functions capture the traffic patterns of road segments and are used to compute the optimal path for a given day and time. Gonzalez et al. [13] exploit the driving and road traffic patterns from historic data in order to consider some non-trivial factors in route planning, such as the experience of local expert drivers. When answering shortest paths queries, their work needs to traverse the road network and hence the query cost is not *bounded*. Also, their routing algorithms only consider the time-varying functions but ignore real-time traffic updates. Our TTP enjoys bounded query cost and responds to live traffic information.

Probabilistic path queries proposed in [16] take traveling time samples for each edge from historic traffic data and construct probability mass distributions for each segment. They leverage the basic conditional probability principle to compute the paths that satisfy a weight (traveling time) requirement l guaranteed by a certain probability τ. The queries require users to specify l or τ. In transportation analysis, it is reasonable for planners to specify τ and find highly reliable paths. However, in online route services, l and τ vary with different SD pairs in practice and the selection of τ is not discussed in [16]. On the contrary, TTP only requires a parameter of k and can be used for both applications.

Another main difference of TTP from time-dependent networks and probabilistic path computation is the way to model historic traffic data. Time-dependent networks model the historic data of each road segment as a *time-varying function* while probabilistic networks represent the historic traffic of roads by different *probability distributions*. TTP regards the traffic data as *high-dimensional cost vectors* and directly exploits them for path computation.

We notice the work of finding route skylines in road networks [18], which is to solve multi-preference path queries and is similar to our Phase I discussed in Section 5.1. However, our problem setting, which is to extract k paths that minimize aggregate travel time error, is totally different.

3 Problem Setting

In this section, we present preliminaries and formulate the TTP problem. The following table summarizes the notation used in our formulation and in the rest of the paper.

Table 1 Notation

Symbol	Meaning
$G(V, E, W_m)$	A directed and multi-weighted network
$v_s(v_d)$	Source (Destination)
(v_i, v_j)	An edge in E
$w_j(e)$	Travel time of $e \in E$ at time instant j
$\mathcal{E}(p)$	The set of edges on path p
$\tau_j(p)$	Travel time of p at time instant j
$P^k_{s,d}$	A k-combination
$\Psi(P^k_{s,d})$	Aggregate value of $P^k_{s,d}$

3.1 Definitions

Definition 1 (Road Network with Historic Traffic) A road network is modeled as a directed and multi-weighted graph $G(V, E, W_m)$, where V is the set of road junctions, E is the set of road segments, and $W_m : E \rightarrow \mathbb{R}^m_+$ is a mapping from edges to m-dimensional cost vectors. Given an edge $e \in E$, we denote its weight vector by $w(e)$ and its travel time at the j-th time instant by $w_j(e)$.

In real world, $w_j(e)$ may correspond to the travel time of a road segment within the j-th time frame, where the length of each time frame could be, e.g., 30 seconds [2] or 15 minutes [5]. We regard this real-world time frame as *time instant* and each of them is indexed by an integer j. In practice, W_m is usually a subset of the entire historic traffic data and is determined by users according to their needs.

Definition 2 (Loop-Free Path) A loop-free path $p = \langle v_{a_1}, v_{a_2}, ..., v_{a_n} \rangle$ is a sequence of distinct nodes such that for $1 \leq i < n$, $(v_{a_i}, v_{a_{i+1}}) \in E$. The start and end nodes on p are denoted by $\mathcal{S}(p)$ and $\mathcal{T}(p)$ respectively. They are also called the source (v_s) and destination (v_d) of p.

For simplicity, we refer to a loop-free path as a path in this paper. The travel time of a path p at time instant j is defined as:

$$\tau_j(p) = \sum_{i=1}^{n-1} w_j((v_{a_i}, v_{a_{i+1}})) = \sum_{e \in \mathcal{E}(p)} w_j(e) \tag{3}$$

where $\mathcal{E}(p)$ denotes the set of edges on path p.

In this paper, we do not consider loops as they incur extra travel time. Our algorithms are designed to avoid loops.

Definition 3 (k-Combination and Aggregate Score) Given a positive integer k, a k-combination $P^k_{s,d} = \{p_1, p_2, ..., p_k\}$ is a set of k paths such that all of them have the same start node v_s and the same end node v_d. The aggregate score of $P^k_{s,d}$ is defined as

$$\Psi(P^k_{s,d}) = \sum_{j=1}^{m} \min_{p \in P^k_{s,d}} \tau_j(p) \tag{4}$$

We illustrate the above concepts on the road network in Figure 2. Each edge is labeled with a cost vector that represents its travel time at 5 (historic) time instants. Table 2 displays all possible paths on the road network and their travel time. For the path $p_1 = \langle v_1, v_2, v_3, v_7 \rangle$, its travel time at time instant 3 is: $\tau_3(p_1) = (7 + 4 + 3) = 14$. Assume that $k = 3$, and consider a 3-combination $P^3_{1,7} = \{p_4, p_5, p_6\}$ for instance. At time instant 1, we have: $\min_{p \in P^3_{1,7}} \tau_1(p) = \min\{19, 17, 15\} = 15$. By summing up the scores over all time instants, we obtain the aggregate score $\Psi(P^3_{1,7})$ is $(15{+}16{+}12{+}21{+}8) = 72$.

Definition 4 (k Traffic-tolerant Paths Query) Given a road network $G(V, E, W_m)$, $v_s, v_d \in V$, and a positive integer k, Traffic-tolerant Paths Query $\mathsf{TTP}(v_s, v_d, k)$ returns a k-combination $P^k_{s,d}$ such that for any possible $P'^k_{s,t}$, $\Psi(P^k_{s,d}) \leq \Psi(P'^k_{s,t})$.

Let us use the road network in Figure 2 for illustration. Table 3 lists all possible 3-combinations and their corresponding aggregates with v_1 as source and v_7 as destination. There are $\binom{6}{3} = 20$ 3-combinations in total. The optimal solution of $\mathsf{TTP}(v_1, v_7, 3)$ is $P_{opt} = \{p_2, p_3, p_4\}$ since its aggregate score $\Psi(P_{opt}) = (16 + 10 + 6 + 14 + 8) = 54$ is the minimum among all combinations.

Table 2 All possible paths from v_1 to v_7, with historic travel time

Path	τ_1	τ_2	τ_3	τ_4	τ_5
$p_1\langle v_1, v_2, v_3, v_7\rangle$	19	20	14	15	16
$p_2\langle v_1, v_4, v_3, v_7\rangle$	18	20	17	**14**	12
$p_3\langle v_1, v_4, v_7\rangle$	16	**10**	**6**	16	14
$p_4\langle v_1, v_5, v_6, v_7\rangle$	19	16	20	21	**8**
$p_5\langle v_1, v_5, v_4, v_3, v_7\rangle$	17	30	23	21	9
$p_6\langle v_1, v_5, v_4, v_7\rangle$	**15**	20	12	23	11

Table 3 All possible 3-combinations $P^3_{1,7}$ and their aggregate scores

$\{p_1, p_2, p_3\}$: 58	$\{p_1, p_4, p_5\}$: 70	$\{p_2, p_4, p_6\}$: 65
$\{p_1, p_2, p_4\}$: 70	$\{p_1, p_4, p_6\}$: 66	$\{p_2, p_5, p_6\}$: 70
$\{p_1, p_2, p_5\}$: 74	$\{p_1, p_5, p_6\}$: 71	$\{p_3, p_4, p_5\}$: 56
$\{p_1, p_2, p_6\}$: 72	$\{p_2, p_3, p_4\}$: **54**	$\{p_3, p_4, p_6\}$: 55
$\{p_1, p_3, p_4\}$: 55	$\{p_2, p_3, p_5\}$: 55	$\{p_3, p_5, p_6\}$: 56
$\{p_1, p_3, p_5\}$: 56	$\{p_2, p_3, p_6\}$: 56	$\{p_4, p_5, p_6\}$: 72
$\{p_1, p_3, p_6\}$: 57	$\{p_2, p_4, p_5\}$: 72	

4 Problem Hardness

In this section, we prove TTP is NP-hard by reduction from the Set-Cover problem, paving the way of devising heuristics in Section 6.

Theorem 1 *The TTP problem is NP-hard.*

Proof We can express an instance of the TTP problem in two equivalent forms: (i) a graph-based instance $\langle G(V, E, W_m), v_s, v_d, k\rangle$, or (ii) a matrix-based instance $\langle k, m, n, \{(i, j, \tau_j(p_i))\}\rangle$, like Table 2. In the latter representation, m represents the number of time instants, n represents the number of possible paths from v_s to v_d in the graph G, and the set $\{(i, j, \tau_j(p_i))\}$ records the travel time of each path p_i at each time instant j, for $1 \leq i \leq n$ and $1 \leq j \leq m$. To facilitate our proof, we consider the decision version of the TTP problem, asking whether there exists a set of k paths $P = \{p_{x_1}, p_{x_2}, \cdots, p_{x_k}\}$, among those n possible paths, such that the score $\sum_{j=1}^{m} \min_{p\in P} \tau_j(p)$ equals to 0.

Next, we present a *reduction scheme* that converts any given instance of the *Set-Cover problem* [10] into a matrix-based instance of the TTP problem. Let $\langle k, CS = \{S_i\}, U\rangle$ be an instance of Set-Cover, where k is an integer, U is a domain set of items, CS is a collection of subsets $S_i \subseteq U$. This problem asks whether there exists a size-k collection $CS' \subseteq CS$ such that they cover all items in U, i.e., $\bigcup_{S_i\in CS'} S_i = U$. The reduction scheme is as follows:

- We set $m = |U|$ and $n = |CS|$. The value of k is the same in both problems.
- Without loss of generality, we rename the items in U as $1, 2, \cdots, m$ (in the Set-Cover instance).
- For each subset $S_i \in CS$ and each item $j \in U$, we set $\tau_j(p_i) = 0$ if $j \in S_i$, or set $\tau_j(p_i) = 1$ otherwise.

There exists a graph-based instance $\langle G(V, E, W_m), v_s, v_d, k\rangle$ with $O(n)$ vertices, as shown in Figure 4, that corresponds to the matrix-based instance $\langle k, m, n, \{(i, j, \tau_j(p_i))\}\rangle$ constructed above. In this graph, the paths connecting v_s and v_d are disjoint.

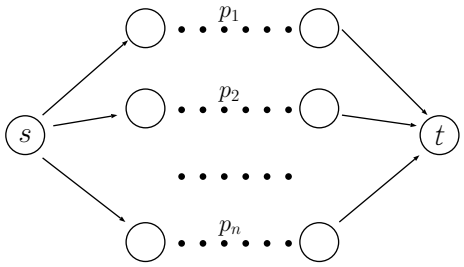


Fig. 4 A graph-based instance corresponding to any given instance of the *Set-Cover problem*

In the subsequent discussion, we focus on the matrix-based instance of TTP only. Note that the size of the constructed TTP instance $\langle k, m, n, \{(i, j, \tau_j(p_i))\}\rangle$ is polynomial to the size of the given Set-Cover instance $\langle k, CS = \{S_i\}, U\rangle$. Also, the construction process takes polynomial time.

Now, we show that a solution CS' of Set-Cover instance corresponds to a solution P of the corresponding TTP instance with aggregate score $\sum_{j=1}^{m} \min_{p\in P} \tau_j(p)$ equal to 0.

We first convert a given $CS' = \{S_{x_1}, S_{x_2}, \cdots, S_{x_k}\}$ to a corresponding $P = \{p_{x_1}, p_{x_2}, \cdots, p_{x_k}\}$ and then derive its aggregate score. By checking the occurrences of items $1, 2, \cdots, m$ in each S_{x_i}, we derive:

$$|\bigcup_{S_{x_i} \in CS'} S_{x_i}| = \sum_{j=1}^{m} \mathcal{B}(\bigvee_{i=1}^{k} (j \in S_{x_i}))$$

where $\mathcal{B}(\cdot)$ is an indicator function that maps `true` to 1 and maps `false` to 0. Since CS' is a solution of Set-Cover, each item $j \in U$ must appear in some S_{x_i}. According to our reduction scheme, the corresponding $\tau_j(p_{x_i})$ in our constructed instance must be 0. Thus, we obtain: $\sum_{j=1}^{m} \min_{i=1}^{k} \tau_j(p_{x_i}) = 0$.

We then convert a given $P = \{p_{x_1}, p_{x_2}, \cdots, p_{x_k}\}$ to a corresponding $CS' = \{S_{x_1}, S_{x_2}, \cdots, S_{x_k}\}$. Since P is a solution, for each time instant j, there exists some path p_{x_i} such that $\tau_j(p_{x_i}) = 0$. According to our reduction scheme, for each item j, there is a corresponding S_{x_i} (in Set-Cover solution) that contains j. Therefore, CS' covers all items in U, and it is a solution of Set-Cover.

Since the Set-Cover problem is NP-hard [10], this proof implies that the TTP problem is also NP-hard. □

It is tempting to adapt existing heuristics for the Set-Cover problem to solve our TTP problem since they are both NP-hard problems. However, in certain cases this does not work. Observe that, the above reduction converts a given Set-Cover instance into to a TTP instance with $\tau_j(p_i)$ as binary values (i.e., either 0 or 1). Existing heuristics for the Set-Cover problem can only be used to solve those TTP instances with binary $\tau_j(p_i)$, but not general TTP instances with $\tau_j(p_i)$ as non-negative real values.

5 Exact Method

In this section, we present an exact method for TTP. Our exact method for TTP (Algorithm 1) consists of two phases: **(Phase I)** generating a set $\mathcal{C}$ of candidate paths and **(Phase II)** finding the optimal combination of k paths from the set $\mathcal{C}$.

A simple implementation is to enumerate all possible paths from v_s to v_d and then examine all size-k combinations of the paths. As an example, we assume $k = 3$ and consider the SD pair (v_1, v_7) in the road network in Figure 2. Since there are $|\mathcal{C}| = 6$ possible paths (from v_1 to v_7) in the road network, we would enumerate $\binom{|\mathcal{C}|}{k} = \binom{6}{3} = 20$ size-3 combinations in total. However, this implementation does not scale well with a large road network.

In the light of this, we optimize the algorithms for both phases to reduce the search space $\binom{|\mathcal{C}|}{k}$. For Phase I, we develop pruning rules to eliminate unpromising paths that cannot contribute to the optimal solution (effectively, to reduce value $|\mathcal{C}|$). For Phase II, we adopt the branch-and-bound paradigm and design pruning rules to discard partial combinations that cannot lead to the optimal solution.

5.1 Phase I: Generating Candidates

We face two challenges in generating candidate paths. First, the number of all possible paths from source to destination is incredibly large on a sizable road network. Exploring all of

Algorithm 1 Exact (Node v_s, Node v_d, Integer k)

1: $\mathcal{C} \leftarrow$ GenerateCandidates (v_s, v_d, k) ▷ Phase I
2: $P_{opt}^k \leftarrow$ FindOptimal (k, $\mathcal{C}$) ▷ Phase II
3: **return** P_{opt}^k

them is impractical. Second, many paths lead to long travel time; such paths could not be included in the optimal solution.

To overcome both challenges, we prune unpromising paths by leveraging the dominance property. Since every edge in the road network has a cost vector $w(e)$ with size m, all possible paths p connecting a SD pair have a m-dimensional cost vector $\overrightarrow{p}$ also. As p corresponds to a vector, we can define and exploit dominance of paths.

Definition 5 (Vector and Path dominance) Let $\overrightarrow{v}$ and $\overrightarrow{u}$ be two m-dimensional vectors. $\overrightarrow{v}$ is said to dominate $\overrightarrow{u}$ if and only if $\forall 1 \leq j \leq m$, $\overrightarrow{v}.j \leq \overrightarrow{u}.j$. We denote this as $\overrightarrow{v} \preceq \overrightarrow{u}$.

Let p and p' be two paths from v_s to v_d. p' is said to dominate p if and only if $\overrightarrow{p'} \preceq \overrightarrow{p}$.

Vector dominance implies the following property, which we will use later.

Lemma 1 (Dominance property) *Given two vectors $\overrightarrow{v}$ and $\overrightarrow{u}$, if $\overrightarrow{v} \preceq \overrightarrow{u}$, then $\sum_{j=1}^{m} \overrightarrow{v}.j \leq \sum_{j=1}^{m} \overrightarrow{u}.j$.*

The concept of path dominance is visualized in Figure 5. A dimension is the travel time of a path at a particular time instant and a point represents a path from v_s and v_d. Every point corresponds to a dominance region. For example, p_3 with $\overrightarrow{p_3} = \langle 3, 4 \rangle$ dominates the points in the blue area , and p_4 with $\overrightarrow{p_4} = \langle 4, 2 \rangle$ dominates the paths in the red area.

Recall that TTP minimizes the aggregate value derived by the paths in a k-combination. Thus, we aim to retain those paths with short travel time over m time slots. By the concept of path dominance, if a path p' dominates another path p, this implies p' has smaller travel time than p at all time instants, so p should be pruned in Phase I. This dominance concept leads to the following pruning rule.

Table 4 Cost vector representations of paths and combinations

Type	Cost Vector $\overrightarrow{v} = \langle \overrightarrow{v}.1, \overrightarrow{v}.2, \cdots, \overrightarrow{v}.m \rangle$
Path p	$\overrightarrow{p} = \langle \cdots, \tau_j(p), \cdots \rangle$
Combination P	$\overrightarrow{P} = \langle \cdots, \min_{p \in P} \tau_j(p), \cdots \rangle$
Prefix path $\hat{p}$	$\overrightarrow{\hat{p}} = \langle \cdots, LB_j(\hat{p}), \cdots \rangle$

Pruning Rule 1 (Path Dominance Pruning) *Given a path p, if there is a path p' such that $\overrightarrow{p'} \preceq \overrightarrow{p}$, then p can be pruned.*

Proof For the sake of discussion, we let $p_1 = p$ and $p'_1 = p'$. Consider a k-combination that contains p_1, say $P = \{p_1, p_2, ..., p_k\}$. Suppose there is a path p'_1 such that $\overrightarrow{p'} \preceq \overrightarrow{p_1}$ and another k-combination is formed $P' = \{p'_1, p_2, ..., p_k\}$.

Consider the j-dimension of the cost vectors and $\overrightarrow{P}$ and $\overrightarrow{P'}$. Observe that $\overrightarrow{P}.j = \min\{\tau_j(p_1), \tau_j(p_2), \cdots, \tau_j(p_m)\}$ and $\overrightarrow{P'}.j = \min\{\tau_j(p'_1), \tau_j(p_2), \cdots, \tau_j(p_m)\}$. Since

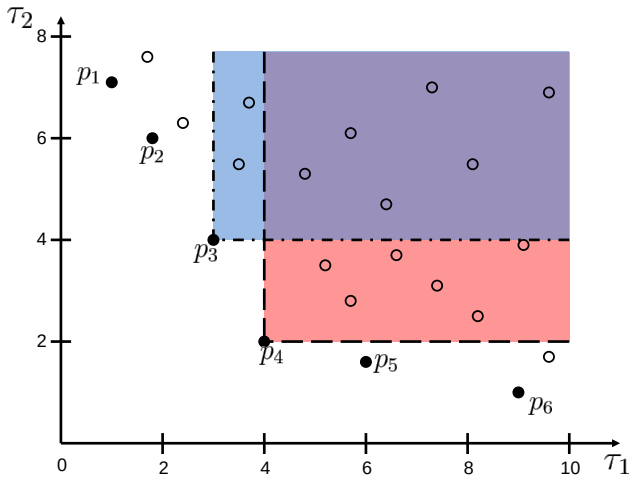


Fig. 5 Concept of path dominance (τ_1 and τ_2 are historic travel time of paths)

$p'_1 \preceq p_1$, we have $\overrightarrow{P'}.j \leq \overrightarrow{P}.j$. By Lemma 1, we derive $\sum_{j=1}^{m} \overrightarrow{P'}.j \leq \sum_{j=1}^{m} \overrightarrow{P}.j$. Thus, $\Psi(P') \leq \Psi(P)$ and the pruning rule is proven. □

This pruning rule is applicable only when all possible paths between v_s and v_d are known. However, enumerating all possible paths is expensive even on a medium-sized road network. Consequently, we aim to avoid exploring the entire search space during enumeration.

Observe that many paths share a common prefix among all possible paths from v_s to v_d. During path enumeration, we can safely disqualify a prefix path (before it reaches v_d) by computing the minimum possible travel time of the paths originating from a common prefix at each time instant j, denoted by $LB_j(\hat{p})$. It is a sum of two terms: (i) the exact travel time of a prefix at time instant j, i.e., $\tau_j(\hat{p})$ and, (ii) the *minimum* travel time from the end node of the prefix to v_d. A prefix path $\hat{p}$ and $LB_j(\hat{p})$ are defined as follows.

Definition 6 (Prefix Path) Given a path $p = \langle v_{a_1}, v_{a_2}, ..., v_{a_n} \rangle$, $\hat{p} = \langle v_{\hat{a}_1}, v_{\hat{a}_2}, ..., v_{\hat{a}_m} \rangle$ is a prefix path of p if and only if $m \leq n$ and for $1 \leq i \leq m$, $v_{a_i} = v_{\hat{a}_i}$.

Definition 7 (Travel Time and Cost Vector of Prefix Path) Given a prefix path $\hat{p}$, for $1 \leq j \leq m$, $LB_j(\hat{p})$ is calculated as

$$LB_j(\hat{p}) = \tau_j(\hat{p}) + \tau_j(sp_j^{last})$$

where $\tau_j(\hat{p}) = \sum_{e \in \mathcal{E}(\hat{p})} w_j(e)$ denotes the exact travel time of a prefix at time instant j and sp_j^{last} denotes the shortest path from the last node of $\hat{p}$ to v_d at time instant j.

The lower-bound cost vector of $\hat{p}$, denoted by $\overrightarrow{\hat{p}}$, is defined as $\overrightarrow{\hat{p}} = \langle LB_1(\hat{p}), LB_2(\hat{p}), ..., LB_m(\hat{p}) \rangle$.

For example, let us consider $\hat{p} = \langle v_1, v_5 \rangle$ and $LB_2(\hat{p})$ in our sample network. We have $\tau_2(\hat{p}) = 8$ and $\tau_2(sp_2^{last}) = 8$, where $sp_2^{last} = \langle v_5, v_6, v_7 \rangle$. Hence, $LB_2(\hat{p}) = (8 + 8) = 16$. By computing $LB_j(\hat{p})$ for $1 \leq j \leq m$, we obtain a cost vector $\overrightarrow{\hat{p}}$ that lower bounds the cost vector of any path sharing the common prefix $\hat{p}$. Similarly, we can apply the dominance concept and the pruning rule for $\overrightarrow{\hat{p}}$.

Pruning Rule 2 (Prefix Path Pruning) *Given a set of paths $\mathcal{D}$ and a prefix path $\hat{p}$, if there is a $p' \in \mathcal{D}$ such that $\overrightarrow{p'} \preceq \overrightarrow{\hat{p}}$, then every path p with the prefix $\hat{p}$ can be pruned.*

Algorithm 2 DepthFirstSearch *implements* GenerateCandidates

```
Algorithm DepthFirstSearch ( Node v_s, Node v_d )
Initialize p̂ ← ⟨v_s⟩
Initialize C ← ∅
Compute τ_j(v, v_d) for each v ∈ V and each time instant j      ▷ by Reverse Dijkstra's algorithm
D ← compute a set of paths by heuristics                       ▷ Section 5.1.1
RecurDFS ( v_s, v_d, p̂, D, C )                                 ▷ candidates stored in C
Add D into C                                                   ▷ for correctness
return C

Algorithm RecurDFS ( Node v_s, Node v_d, Path p̂, Path set D, Path set C )
u_last ← T(p̂)
if u_last ≠ v_d then
    if ∀p ∈ D, p⃗ is not dominated by p̂⃗ then                    ▷ pruning rule
        for each node v adjacent to u_last do
            if v ∉ p̂ then
                append v to p̂
                RecurDFS ( v_s, v_d, p̂, D, C )
                remove v from p̂
else
    C ← C ∪ {p̂}
```

We present the implementation of Phase I in Algorithm 2. First, we initialize an empty prefix path and an empty candidate set $\mathcal{C}$. The former is to store the currently expanded prefix path while the latter aims to maintain the candidates found. Next, we compute the shortest path distances from $v \in V$ to v_d for each time instant j and apply a heuristic to find a set of pruning paths $\mathcal{D}$. All of them will be used to support our pruning rules. We will discuss how to select the set $\mathcal{D}$ shortly (Section 5.1.1). Then, we apply a DFS-like procedure to find all qualified candidates. The procedure recursively constructs different prefixes stemming from v_s and compares them against $\mathcal{D}$. If a prefix is dominated by a path in $\mathcal{D}$, the algorithm discards and stops expanding the prefix. Otherwise, the recursion keeps expanding the prefix by visiting the neighbors of its end node until reaching v_d. Each completely expanded path (from v_s to v_d) becomes a candidate for the next phase. Finally, $\mathcal{D}$ is added to the candidate set $\mathcal{D}$ for correctness. Notice that it is possible that Algorithm 2 could produce only one candidate, implying that the candidate has been the shortest path in the entire traffic history.

5.1.1 Selection of the Pruning Path Set $\mathcal{D}$

In the previous section, we mentioned that every path p corresponds to distinct dominance regions. The selection of $\mathcal{D}$ is critical for pruning efficiency. Including too many paths in $\mathcal{D}$ may actually cripple performance because pruning becomes too slow (we need to check every element against a prefix path one by one). We have empirically found that $\mathcal{D}$ yields the best performance in most cases when it includes the $k + 1$ paths described below. Note that this does not affect correctness, but only determines the trade-off between the effectiveness of pruning and its processing overhead.

We first introduce a new notation of edges, $w_s(e)$. It represents the weight of an edge dedicated to shortest path search and is used to find $\mathcal{D}$. $w_s(e)$ is introduced in order not to

mix with $w_j(e)$. Initially, for every edge, we add up the travel time of the edge of all m time instants, i.e., $w_s(e) = \sum_{j=1}^{m} w_j(e)/m$. Then, we compute the shortest path between v_s and v_d using $w_s(e)$ and insert it into $\mathcal{D}$. The intuition is that a path is not bad at all time instants.

Next, in the road network, we set the weight of each edge e to $w_s(e) = \min_{j=1}^{m} w_j(e)$. We then execute the following steps for k iterations. In each iteration, we perform a shortest path search from v_s to v_d, obtain a shortest path sp_i, and insert it into $\mathcal{D}$. Then, we update $w_s(e) = \max_{j=1}^{m} w_j(e)$ for each edge e on sp_i. We hope that this would force the shortest path search in subsequent iterations to find other paths that are significantly different from sp_i.

5.2 Phase II: Finding Optimal Combination

In this section, we present the implementation for Phase II, i.e., enumerating k-combinations of paths from the candidate set $\mathcal{C}$ in order to find the optimal k-combination. Since we have obtained the cost vectors of candidate paths in Phase I, the road network is no longer required in this phase.

The number of k-combinations of paths is $\binom{|\mathcal{C}|}{k}$, so it is expensive to generate them, especially when $|\mathcal{C}|$ is large. Thus, we develop pruning rules to prevent exploring unnecessary k-combinations. The key idea of the pruning rules is to early stop extending any *partial combination* $\hat{P}$ (which contain fewer than k candidates) by computing its lower bound aggregate value.

Continuing with our running example, Table 3 lists out all 3-combinations and their aggregate travel time errors. Observe that some of the combinations share common paths. For example, $\{p_1, p_2, p_3\}$, $\{p_1, p_2, p_4\}$, $\{p_1, p_2, p_5\}$ and $\{p_1, p_2, p_6\}$ have $\{p_1, p_2\}$ as common paths. Suppose that $P_{best} = \{p_1, p_2, p_3\}$ is the best combination found so far and $\Psi(P_{best})$ is 58. Let us consider a partial combination $\hat{P} = \{p_1, p_2, p_*\}$, where $\{p_1, p_2\}$ are *fixed paths* $\hat{P}_f$ and p_* is a *variable* path selected from $\hat{P}_v = \{p_4, p_5, p_6\}$. We can derive the lower bound aggregate value of $\hat{P}$, denoted by $\Psi_{lb}(\hat{P})$, using Definition 8. $\Psi_{lb}(\hat{P})$ is calculated as $\sum_{j=1}^{m} \min\{\min_{p\in\{p_1,p_2\}} \tau_j(p), \min_{p_*\in\{p_4,p_5,p_6\}} \tau_j(p_*)\} = 15 + 16 + 12 + 14 + 8 = 65$ according to Table 3. Since $\Psi_{lb}(\hat{P}) > \Psi(P_{best})$, we can safely discard three combinations derived from $\hat{P}$, namely $\{p_1, p_2, p_4\}$, $\{p_1, p_2, p_5\}$ and $\{p_1, p_2, p_6\}$, by Pruning Rule 3.

Definition 8 (Lower Bound Aggregate Value of Partial k-combination) Given a partial k-combination $\hat{P}$ with its fixed path set $\hat{P}_f$ and variable path set $\hat{P}_v$, $\Psi_{lb}(\hat{P})$ is defined as

$$\sum_{j=1}^{m} \min\{\min_{p\in\hat{P}_f} \tau_j(p), \min_{p_*\in\hat{P}_v} \tau_j(p_*)\} \tag{5}$$

With the above lower bound cost equation, we derive the following pruning rule.

Pruning Rule 3 (Partial k-combination Pruning) *Let P_{best} be a known k-combination. Given a partial combination $\hat{P}$, if $\Psi_{lb}(\hat{P}) > \Psi(P_{best})$, then the derived combinations of $\hat{P}$ can be pruned.*

5.2.1 Efficient computation of $\Psi_{lb}(\hat{P})$ by $\mathcal{TC}$ matrix

It is expensive to compute $\Psi(\hat{P})$ by Definition 8 directly because the term $\min_{p_* \in \hat{P}_v} \tau_j(p_*)$ takes $O(|\hat{P}_v|)$ computation time. We propose a structure called $\mathcal{TC}$ matrix to support retrieving term $\min_{p_* \in \hat{P}_v} \tau_j(p_*)$ in constant time.

The $\mathcal{TC}$ matrix is a $n \times m$ matrix, where m is the number of time instants and n is the size of $\mathcal{C}$. Its entries are used to store the lower bound travel time of some variable path sets, i.e., $\min_{p_* \in \hat{P}_v} \tau_j(p_*)$ in Equation 5. Let $\mathcal{C} = \{c_1, c_2, ..., c_n\}$ and the entry $\mathcal{TC}[i, j]$ be defined as the lower bound travel time of $c_i, ..., c_n$ at the j-th time instant:

$$\mathcal{TC}[i, j] = \min_{x=i}^{n} \tau_j(c_x) \tag{6}$$

Table 5 illustrates the $\mathcal{TC}$ matrix for the paths in Table 2. Intuitively, the i-th row of $\mathcal{TC}$, denoted by $\mathcal{TC}[i, \cdot]$, is the lower bound cost vector of the candidates $c_i, ..., c_n$ and supports our partial k-combination pruning. By constructing the $\mathcal{TC}$ matrix incrementally in descending order of i, we can achieve $O(n \cdot m)$ construction time.

Next, by using $\mathcal{TC}$ matrix in Table 5, we illustrate how to compute the lower bound cost of a partial combination $\hat{P}$. Suppose $k = 3$ and consider $\hat{P}$ with $\hat{P}_f = \{p_4, p_5\}$ and $\hat{P}_v = \{p_6, p_2, p_1\}$. To calculate $\Psi_{lb}(\hat{P})$, we first fetch the cost vectors of p_4, p_5 from Table 2, which are $\langle 19, 16, 20, 21, 8\rangle$ and $\langle 17, 30, 23, 21, 9\rangle$. Next, we fetch the lower bound cost vector $\langle 15, 20, 12, 14, 11\rangle$ of $\hat{P}_v$, i.e., $\mathcal{TC}[3, \cdot]$, from Table 5. Note that we need to compute this lower bound cost vector from scratch if we do not maintain $\mathcal{TC}$ matrix. Then, we combine these three cost vectors by taking the minimum value in each dimension to obtain $\langle 15, 16, 12, 14, 8\rangle$, and calculate the lower bound cost as: $\Psi_{lb}(\hat{P}) = (15 + 16 + 12 + 14 + 8) = 65$. If $P_{best} = \{p_2, p_3, p_4\}$ with $\Psi(P_{best}) = 54$, we do not examine all 3-combinations derived from $\hat{P}$ (i.e., $\{p_4, p_5, p_6\}$, $\{p_4, p_5, p_2\}$ and $\{p_4, p_5, p_1\}$) since $\Psi(P_{best}) < \Psi_{lb}(\hat{P})$.

Table 5 $\mathcal{TC}$ matrix for paths in Table 2, ordered by $\tau_{min}(p)$

Content of $\mathcal{C}$	**Path**	τ_1	τ_2	τ_3	τ_4	τ_5
c_1	p_3	15	10	6	14	8
c_2	p_4	15	16	12	14	8
c_3	p_5	15	20	12	14	9
c_4	p_6	15	20	12	14	11
c_5	p_2	18	20	14	14	12
c_6	p_1	19	20	14	15	16

5.2.2 Enumeration Algorithm

We adopt the *branch-and-bound* paradigm together with partial k-combination pruning to enumerate the optimal solutions efficiently. Algorithm 3 shows the pseudocode for enumerating path combinations and finding the optimal k-combination P_{opt} with the aid of $\mathcal{TC}$ matrix. First, we compute the $\mathcal{TC}$ matrix of $\mathcal{C}$. Next, we execute *RecurBranch* recursively to insert the i-th candidate c_i into $\hat{P}$. If $\hat{P}$ contains fewer than k paths, then we compare its lower bound cost ($\Psi_{lb}(\hat{P})$) with that of the best combination found so far (stored in P_{opt}). If $\hat{P}$ has a smaller cost, then we call *RecurBranch* to further expand it. When $\hat{P}$ contains k paths, we can compute its exact cost and check whether it is better than the current P_{opt}.

The effectiveness of partial combination pruning depends on how *early* we can obtain a complete k-combination with a high pruning power, i.e., close to the optimal solution. In order to achieve early discovery, we insert the candidates in ascending order of $\tau_{min}(p) = \min_{j=1}^{m} \tau_j(p)$ to $\hat{P}$ as shown in Table 5 and Algorithm 3 since this order intuitively allows early discovery of a good k-combination.

Theorem 2 *The time and space complexities of Algorithm 3 are $O(m \cdot \binom{|\mathcal{C}|}{k})$ and $O(|\mathcal{C}| \cdot m)$ respectively.*

Proof In the worst case, if the pruning rules do not work, Algorithm 3 enumerates all $\binom{|\mathcal{C}|}{k}$ combinations and takes $O(m)$ time to compute Ψ per combination. Therefore, it takes $O(m \cdot \binom{|\mathcal{C}|}{k})$ time. We need to maintain the cost vectors of the candidates (e.g., Table 2) and their corresponding $\mathcal{TC}$ matrix. Both of them require $O(|\mathcal{C}| \cdot m)$ space and hence Algorithm 3 requires $O(|\mathcal{C}| \cdot m)$ space too. □

Theorem 3 *The exact algorithm yields the optimal solution of TTP.*

Lemma 2 *Phase I of exact algorithm preserves all non-dominated paths.*

Proof We prove this lemma by contradiction. Assume Phase I of exact algorithm does not preserve all non-dominated paths, i.e., a non-dominated path $p' \in \bar{\mathcal{D}}$ is pruned. By Pruning Rule 1, if p' is pruned, there exists another path p'' such that $p'' \preceq p'$. This contradicts $p' \in \bar{\mathcal{D}}$. Thus, the proof is completed. □

Lemma 3 *Phase II of exact algorithm yields the optimal solution of TTP.*

Proof In the worst case, Phase II computes all path combinations formed by the candidates $\mathcal{C}$ in Phase I. By Pruning Rule 3, all disqualified combinations can be discarded by the best combination found so far P_{best} and P_{best} will be the optimal solution finally. Hence, the exact algorithm yields the optimal solution of TTP and the proof is completed. □

6 Heuristic Methods

Since our problem is NP-hard, our proposed exact algorithm may incur high running time in the worst case. In this section, we present two heuristics to bound the number of candidates $|\mathcal{C}|$ and thus reduce the computation cost. First, we propose a method (TP) that reduces the cost of candidate generation by a heuristic (Phase I). Second, we develop a method (ATP) that makes 'best-effort' to find a low-cost solution within a given time limit.

6.1 Top-Picker Algorithm

In our exact algorithm, the candidate set $\mathcal{C}$ is large even if we apply pruning rules in Phase I. This leads to a huge number of path combinations in Phase II. In order to reduce the total computation cost, Top-Picker Algorithm (TP) generates a bounded number of candidates by a heuristic (Phase I) and reuses the combination enumeration of the exact method (Phase II).

The idea of TP is to limit the size of the candidate set $\mathcal{C}$, say, to at most m. It computes the shortest path sp_j (from v_s to v_d) at each time instant, and then inserts these paths into the candidate set $\mathcal{C}$.

Algorithm 3 BranchEnumerate *implements* FindOptimal

Algorithm BranchEnumerate (Integer k, Paths $\mathcal{C}$)
1: Initialize $P_{opt} \leftarrow \varnothing$ ▷ optimal solution in P_{opt}
2: $\mathcal{C}^s \leftarrow$ sort $\mathcal{C}$ in ascending order of τ_{min}
3: Compute the matrix $\mathcal{TC}$ by using $\mathcal{C}^s$
4: **for** $i \leftarrow 1$ to $|\mathcal{C}^s| - k + 1$ **do**
5: $\quad \hat{P} \leftarrow \{c_i^s\}$
6: $\quad$ RecurBranch (k, $\hat{P}$, $\mathcal{C}^s$, $\mathcal{TC}$, P_{opt})
7: **return** P_{opt}

Algorithm RecurBranch (Integer k, Path set $\hat{P}$, Path set $\mathcal{C}^s$, Matrix $\mathcal{TC}$, k-combination P_{opt})
1: **if** $|\hat{P}| < k$ **then**
2: $\quad z \leftarrow$ largest index of candidates in $\hat{P}$
3: $\quad$ **for** $i \leftarrow z + 1$ to $|\mathcal{C}^s| - (k - |\hat{P}|) + 1$ **do**
4: $\quad\quad \hat{P} \leftarrow \hat{P} \cup \{c_i^s\}$
5: $\quad\quad$ **if** $\Psi_{lb}(\hat{P}) < \Psi(P_{opt})$ **then** ▷ by using $\mathcal{TC}$
6: $\quad\quad\quad$ RecurBranch (k, $\hat{P}$, $\mathcal{C}^s$, $\mathcal{TC}$, P_{opt})
7: $\quad\quad \hat{P} \leftarrow \hat{P} \setminus \{c_i^s\}$
8: **else**
9: $\quad P \leftarrow \hat{P}$ ▷ becomes a k-combination
10: $\quad$ **if** $\Psi(P) < \Psi(P_{opt})$ **then**
11: $\quad\quad P_{opt} \leftarrow P$

Let us use Table 2 as an example with $k = 3$. First, we find the shortest paths at each time instant, which are p_6, p_3, p_3, p_2, p_4, respectively. Then, we insert these paths into the candidate set $\mathcal{C} = \{p_2, p_3, p_4, p_6\}$. In Phase II, we apply Algorithm 3 to enumerate all 3-combinations of $\mathcal{C}$, such as $\{p_2, p_3, p_4\}, \{p_2, p_3, p_6\}, \{p_2, p_4, p_6\}, \{p_3, p_4, p_6\}$. Finally, we compute their costs and return $P_{opt} = \{p_2, p_3, p_4\}$.

Theorem 4 *The time complexity of Top-Picker Algorithm is* $O(m \cdot (D + \binom{m}{k}))$ *where* D *is the time complexity of the shortest path algorithm. Its space complexity is* $O(\max\{|V| + |E|, m^2\})$.

Proof In Phase I, TP executes shortest path search m times to find the candidate set $\mathcal{C}$, resulting in time complexity $O(m \cdot D)$, where D is the time complexity of the shortest path algorithm. Since the size of $\mathcal{C}$ is at most m, Phase II may examine at most $\binom{m}{k}$ path combinations and take $O(m)$ to derive Ψ. As a result, the time complexity of Top-Picker Algorithm is $O(m \cdot (D + \binom{m}{k}))$.

We need to maintain the road network for shortest path search in Phase I, which requires $O(|V|+|E|)$ space. In Phase II, we can solely maintain the cost vectors of all candidates and $\mathcal{TC}$ matrix, which requires $O(m^2)$ space. Therefore, its space complexity is $O(\max\{|V| + |E|, m^2\})$. □

6.2 Anytime Top-Picker Algorithm

In some applications, the query user (e.g., transportation planner) is fine with an approximate solution and specifies a time limit T_{limit} for finding the solution. To support this requirement, we propose the Anytime Top-Picker Algorithm (ATP), which attempts to find a good approximate solution for TTP within a given time limit.

Algorithm 4 GeneratePaths-TP (Node v_s, Node v_d)

```
1: Initialize C ← ∅
2: for j ← 1 to m do
3:     sp_j ← the shortest path from v_s to v_d at instant j
4:     if sp_j is not in C then
5:         C ← C ∪ {sp_j}
6: return C
```

In order to allocate the time fairly on (i) finding candidates and (ii) enumerating path combinations, we interleave phases I and II in this algorithm. For this purpose, we implement an incremental function for Phase I, called GetNextSP(), as shown in Algorithm 5. It iteratively returns distinct shortest paths across the given m time instants until exhausting all of them. Algorithm 6 shows the pseudocode of ATP. Initially, ATP fetches k distinct shortest paths over m instants and stores them into a candidate set $\mathcal{C}$. These k paths are used to initialize the best combination found so far P_{opt}. If there is time left, then it retrieves another shortest path sp. Subsequently, a new k-combination P is formed by combining sp with each $(k-1)$-combinations of $\mathcal{C}$, and replaces the current optimal solution if it has a better score. Upon reaching the time limit or exhausting all shortest paths, the algorithm returns the best combination found so far P_{opt} as the result.

Algorithm 5 GetNextSP (Node v_s, Node v_d, Hash Table $\mathcal{H}$)

```
1: for each unscanned instant j do
2:     sp ← the shortest path from v_s to v_d at instant j
3:     if sp is not in H then
4:         Insert sp to H
5:         return sp
6: return ∅
```

Algorithm 6 AnytimeTP (Node v_s, Node v_d, Integer k, Time T_{limit})

```
1: C ← ∅, P_opt ← ∅
2: Initialize a hash table H
3: for i ← 1 to k do
4:     sp ← GetNextSP ( v_s, v_d, H )
5:     C ← C ∪ {sp}
6: P_opt ← C
7: while sp ← GetNextSP ( v_s, v_d, H ) do
8:     for each (k−1)-combination P^{k−1} of C do
9:         if T_limit is used up then
10:            return P_opt
11:        P_cur ← P^{k−1} ∪ {sp}
12:        if Ψ(P_cur) < Ψ(P_opt) then
13:            P_opt ← P_cur
14:    C ← C ∪ {sp}
15: return P_opt
```

We illustrate how ATP works in the example in Table 2. Assume $k = 3$ again. Table 6 shows the detailed execution steps of ATP. First, we fetch $k = 3$ paths (i.e., p_6, p_3, p_2) and

insert them into the candidate set $\mathcal{C}$. Next, we initialize $P_{opt} = \mathcal{C} = \{p_6, p_3, p_2\}$, whose score is $\Psi(P_{opt}) = 55$. If there is time left, then we fetch the next distinct shortest path (p_4). Then, we enumerate all 2-combinations of $\mathcal{C}$, such as $\{p_6, p_3\}$, $\{p_3, p_2\}$ and $\{p_6, p_2\}$, and combine each of them with p_4 to form a new 3-combination. If a new combination has a smaller score, then it becomes the current optimal result. Finally, the algorithm reports $\{p_2, p_3, p_4\}$ with $\Psi = 54$ as the answer.

Table 6 Execution steps of ATP on the example in **Table 2**

Procedure	Checked $P^3_{1,7}$	$\Psi(P^3_{1,7})$	Current P_{opt}
Initialize	$\{p_6, p_3, p_2\}$	55	$\{p_6, p_3, p_2\}$
GetNextSP : p_4	$\{p_4\} \cup \{p_6, p_3\}$	55	$\{p_6, p_3, p_2\}$
	$\{p_4\} \cup \{p_6, p_2\}$	65	$\{p_6, p_3, p_2\}$
	$\{p_4\} \cup \{p_3, p_2\}$	**54**	$\{p_2, p_3, p_4\}$

7 TTP in Practice

7.1 Selection Policy

Up to the previous section, we implicitly assume that we are given a set of traffic data to compute TTPs. In practice, the set of traffic data may be extracted from a large traffic data repository. For example, users (e.g., transportation planners and analysts) are interested in only a small portion of the data (e.g., the most recent) since the volume of the entire traffic data is huge.

Therefore, in this section, we discuss some guidelines for selecting a subset of traffic data. We call the set of guidelines a *selection policy* which is suited to the *application scenarios* we discussed in Section 1.1. It contains two parameters, namely the number of days (D) and the number of time instants per day (L). The following discussion on selection policy is in the context of *online route services*. However, the selection policy for transportation analysis can be defined similarly.

Intuitively, D determines how many historic traffic records are used for pre-computation. Including all traffic recorded several years ago from the time of recurrent route queries may not be useful since the traffic patterns may change over time. In contrast, only considering the traffic data collected few weeks or months ago from the time of the queries may be more meaningful as they reflect the recent traffic of the road network.

Besides, the collection period of historic traffic data is equally important. If the journey starts in the morning, it is not reasonable to include the traffic data recorded at midnight since the traffic conditions during these two periods are expected to be different. Additionally, we expect that there is a periodicity in traffic and the traffic at the same 'hours' on different dates has similar patterns. For example, the traffic between 8:00am and 9:00am on weekdays in a month is expected to be similar since the majority of citizens go for work in these peak hours and the traffic in this period may be useful to a route which starts, say, at 8:00am. We consider this daily time window for path pre-computation as another selection parameter (L) - the number of time instants per day.

Using our approach for short periods is more suitable for online route services since historic traffic data over short periods reveal limited short-term fluctuation in travel time of road segments. In contrast, using historic traffic data with too many time frames introduces

much fluctuation and may weaken the robustness of TTPs as they need to take them into account while the number of paths remains unchanged.

The policy we recommend is a simple but intuitive policy to extract a subset of historic traffic data. It can be further extended to take *weekday* patterns into account. This can be realized by introducing an additional parameter (W) which determines if the traffic of a particular weekday is considered. For instance, if users are interested specifically in Friday traffic at 18:00-19:00, they can extract the historic traffic that matches the criteria in the past few months and apply it to TTP. With rich knowledge on the road network and traffic of their cities, system administrators or transportation analysts can highly customize the selection policy and hence exploit meaningful historic traffic data for traffic-tolerant paths computation.

To sum up, the selection policy determines a subset of historic traffic data for pre-computation. We are going to evaluate the effect of the two parameters on the accuracy of TTPs in our experiments (Chapter 8). However, the art of designing and customizing an appropriate selection policy is orthogonal to our approach.

7.2 TTP in Online Route Services

In this paper, we emphasize on recurrent queries (e.g., finding the fastest route from home to office at 8:00am every day). The source and destination can be identified from users' query history or it can be pre-defined by users. For example, Google Now allows users to specify home and workplace, and reports the fastest route to the users every morning. It is also able to discover users' frequently visited places and provide corresponding navigation suggestions. In the context of recurrent queries, TTPs act as candidate paths. We recommend to update TTPs offline periodically (e.g., weekly) in order to reflect the recent traffic patterns of road networks. The update frequency can be determined by system administrators. As the source and destination of recurrent queries are given, it is not necessary to decide how to select the pairs of query locations and how many such pairs should be selected.

However, our TTP framework can be extended in order to answer online ad-hoc route queries. A straightforward adaption of TTP to cope with situations like sudden changes in traffic conditions (e.g., traffic accidents, protests, and congestions) is that we check whether there are any segments on TTPs suffering serious congestions and then we replace those congested segments with other segments offering shorter travel time in an online phase.

For instance, we have a traffic-tolerant path $p = \langle v_s, ..., v_u, ..., v_w, ..., v_t \rangle$ and find that the segment $s_{old} = \langle v_u, ..., v_w \rangle$ is undergoing a traffic jam. We search for the fastest path between v_u and v_w (sp) based on the current traffic and replace s_{old} with sp. Since v_u and v_w are close to each other, the cost of fastest path search is small. This adaption not only preserves the quality of traffic-tolerant paths but also maintains reasonable query time.

Another possible extension can also be used to answer ad-hoc route queries. This extension is divided into 2 phases, namely *offline* and *online* phases. In *offline* phase, we divide the road network into partitions. Each partition has border nodes (i.e., nodes that are adjacent to some node in another partition); any path between two different partitions must pass through border nodes of the partitions. Therefore, we can pre-compute k traffic-tolerant paths between all pairs of border nodes. In *online* phase, when an ad-hoc route query comes, we execute forward and backward Dijkstra from source and destination respectively to their nearest borders based on live travel time. Then, we fetch the k pre-computed traffic-tolerant paths between those two border nodes and hence form k candidate paths from source to

destination. The path with the minimum travel time is reported to users. Note that the two adaptations can be combined to answer route queries in case there is a serious traffic jam.

8 Experimental Evaluation

8.1 Road Network and Traffic Data

Table 7 lists the information of road networks used in our experiments. They are United Kingdom (**UK**) and Colorado (**COL**), which are available at [3] and [1] respectively.

For UK, we downloaded real and historic traffic data from [5] from January to March of 2013. Each traffic record is linked to a road segment in UK based on the unique identifiers of roads. The traffic data were recorded every 15 minutes and hence there are 96 traffic records per day for each road segment.

Since the historic traffic data of COL are *not* available, we synthetically generate traffic data in the following way. We generate the travel time for m time instants. At each time instant j, we pick a random number from $\{-1, 1\}$ as $sign_j$ to determine whether to *increase* or *decrease* the travel time of all edges. Then, for each edge, we select a random number from $[0..X]$ and $w_j(e)$ is calculated as $\frac{l(e)}{s} \times (1 + sign_j \cdot X\%)$, where $l(e)$ is the road segment length and s is the vehicle speed. We set s as 60km/h in our experiments. The values of X are shown in Table 8.

8.2 Experimental Setup

In the introduction, we suggest two applications of our TTP problem, namely transportation planning and online route services. The former requires fast computation time, whereas the latter focuses on the travel time error of the pre-computed paths against real-time traffic. Thus, in our experiments, we measure the computation time and travel time error.

Training and testing sets of traffic data: In order to evaluate the travel time error of the methods, we divide the traffic data into *training* and *testing* sets. We take the training set for computing TTP solutions, and take the testing set for testing the travel time error of such solutions.

For UK, training and testing sets of traffic data are governed by the equation $m = D \times L$, where D is the number of training (testing) days and L is the number of training (testing) time instants per day. Testing sets contain traffic data collected after the data in the training set. L specifies the granularity of time periods, (e.g., in UK, a time period 08:00-09:00 implies L = 4 since the traffic data of UK are recorded every 15 minutes).

In our experiments, we vary D and L of training sets. We fix D of the testing set as traffic data collected during 16 - 31 March, 2013 (i.e., D = 16) but vary L according to L of the training sets. For instance, we take the traffic data during 08:00-08:30 (rush hour) and 15 days before the testing period for training. Then, we use the traffic data during 08:00-08:30 in the testing period for testing.

For COL, we simply generate two sets of synthetic traffic data for each network according to Section 8.1. One set is for training while another set is for testing. The number of time instants (m) for training and testing is shown in Table 8.

Average travel time error: Let P be the path combination, computed by a method, by using the training set. We measure the travel time error of P by using the testing set Q

as follows. The error of P, for a given pair of query $q_i \in Q$ and testing timestamp j, is measured as:

$$\epsilon_j^{q_i} = \left(\min_{p \in P} \tau_j(p)\right) - \tau_j(sp_j)$$

where $\tau_j(p)$ is the travel time of path p at timestamp j and sp_j is the shortest path for q_i at timestamp j. In our experiments, we report the average travel time error ϵ_{avg}^Q of P as:

$$\epsilon_{avg}^Q(P) = \texttt{AVG}\{\epsilon_j^{q_i} : i = 1..|Q|, j = 1..m\}$$

where m is the number of (distinct) time instances in Q. Note that this equation resembles Equation 1.

Alternative error measures: We also report alternative error measures in some representative experiments. To generalize the above travel error, we replace the average function with a statistical function $\mathcal{G}$ to obtain:

$$\epsilon_{\mathcal{G}}^Q(P) = \mathcal{G}\{\epsilon_j^{q_i} : i = 1..|Q|, j = 1..m\}$$

Examples of $\mathcal{G}$ include: (i) the minimum, (ii) the 25-th percentile, (iii) the median, (iv) the 75-th percentile, (v) the maximum. We denote their corresponding travel time errors as: ϵ_{min}^Q, $\epsilon_{25\%}^Q$, $\epsilon_{50\%}^Q$, $\epsilon_{75\%}^Q$, ϵ_{max}^Q, respectively.

In addition, we calculate the frequency of real fastest path(s) appearing in P as:

$$SPfreq(P) = \frac{\sum_{i=1}^{|Q|} \sum_{j=1}^{m} \mathcal{B}_j^{q_i}}{m|Q|} \cdot 100\%$$

where $\mathcal{B}_j^{q_i}$ returns 1 if P contains the fastest path for query q_i at timestamp j, and $\mathcal{B}_j^{q_i}$ returns 0 otherwise.

Query generation: We adopt two query generation methods in our experiments, namely (1) *uniformly random* query and (2) *distance threshold* query. The former picks 100 source-destination pairs uniformly at random and the *average* time errors of these pairs are reported in the figures in the subsequent sections.

The latter is used to evaluate the effect of distance on travel time error. It picks a source at random and runs Dijkstra's algorithm until reaching a distance threshold (δ_t). We choose the first node whose shortest distance from the source exceeds δ_t as the destination. For each road network, we select N thresholds and N groups of queries as follows. We pick a random source and compute its largest shortest path distance by Dijkstra's algorithm (i.e., $\max_{v \in V} \delta(s, v)$ where $\delta(\cdot, \cdot)$ denotes the shortest path distance between two nodes). Then, we compute the average largest shortest path distance, denoted by δ_t^{max}, of 100 random sources. For the i-th group (where $1 \leq i \leq N$), we define its distance threshold as $\delta_t^i = i \cdot \frac{\delta_t^{max}}{(N+1)}$, and generate 100 source-destination pairs for the group. Then, we report the average travel time error of each group. We set $N = 5$ in our experiments.

For the value of k, we test k from 1 to 10 and choose $k = 5$ by default, which is the same as [21].

Methods and competitors: Our proposed methods are: the exact method (which is shown as TTP in figures) and two heuristic methods (TP and ATP). Our competitors are two representative heuristic methods proposed in [21]: *K-variance* (K-VAR) and *Y-moderate* (Y-MOD).

The idea of K-VAR is to maintain a Gaussian distribution $\mathcal{N} \sim (\mu, \sigma^2)$ for the historic traffic data for each edge. To find a candidate path, it samples the travel time from the distribution for each edge and computes a shortest path from v_s to v_d. It repeats the travel time

sampling and the shortest path search after a fixed number of iterations in order to find k candidate paths. Y-MOD is a variant of Yen's algorithm [26] which computes k loop-free shortest paths. Its idea is to guarantee that the resulting k candidate paths have a fixed fraction f of overlapping edges. Any additional parameters in these two methods are configured according to [21].

We implement all methods in C++ and evaluate them in subsequent experiments. All the experiments were run on a PC with 3.4 GHz Intel® Core™ i7 CPU and 8 GB RAM in Linux environment.

Table 7 Road network size

Network	#Nodes	#Edges	Traffic
United Kingdom (**UK**)	2,321	4,996	Real [5]
Colorado (**COL**)	435,666	1,042,400	Synthetic

Table 8 Experiment parameters

Parameters	Values	Default
k	1,2,...,10	5
No. of days (D)	15,30,45,60	15
No. of time instants per day (L)	1,2,3,4	1
m for synthetic traffic	15,30,45,60	30
$X\%$ for synthetic traffic	5,10,15,20	10

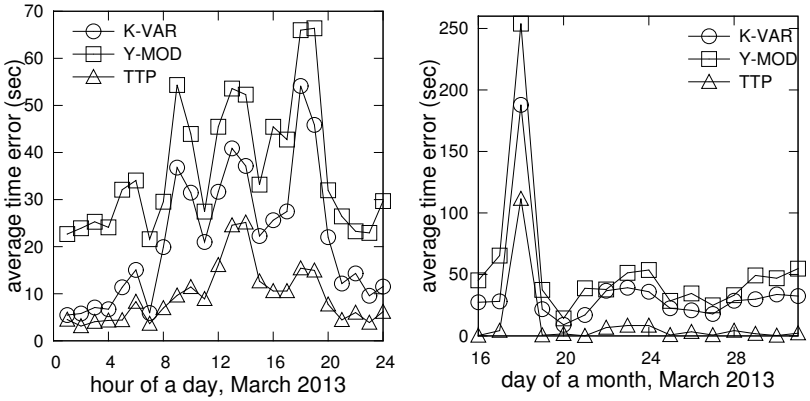
8.3 Real Traffic Data

In this section, we present the travel time error and the efficiency of TTP in various perspectives including (i) different hours of a day, (ii) different days of a month, (iii) increasing the number of paths (k), (iv) increasing the number of time instants (m), and (v) increasing distance from sources. Uniformly random queries are used in (i) to (iv) while distance threshold queries are used in (v). In this section, TTP in the figure refers to the results generated by the exact algorithm.

Different hours of a day: First, we evaluate how the average time error varies with different time of a day since this reveals some traffic patterns of UK. Figure 6(a) shows the average time errors of TTP and two competitors at different hours of a day. TTP achieves a smaller time error throughout the day and outperforms the others especially at the rush hours (i.e., 08:00 and 17:00) by at least 3 times. Although there is a spike at 13:00-14:00 for TTP its time error is still smaller than that of others. The figure also reveals that the traffic of UK fluctuates in three periods, namely 08:00-09:00, 13:00-14:00 and 17:00-18:00. The default time period in the subsequent experiments is 08:00-09:00.

Different days of a month: Figure 6(b) shows the average time errors of three algorithms across different testing days of March, at 08:00-09:00. TTP has consistently smaller errors than the others. Although all of them suffer from a sudden rise in time error on 18 March, 2013, TTP can still obtain a much lower average time error of about 2 minutes, while K-VAR and Y-MOD have average time errors of about 3 minutes and 4 minutes respectively.

Varying k: As shown in Figure 7(a), the average time errors of all algorithms diminish with increasing k and start to converge when $k > 5$. This is because including more paths



(a) Varying time of a day (b) Varying day of a month

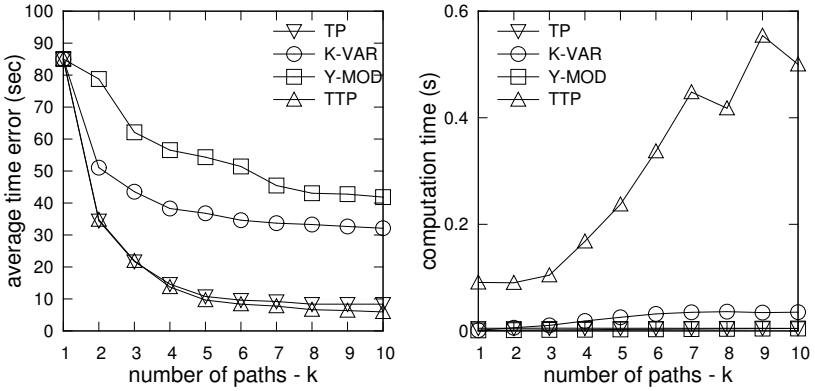
Fig. 6 $k = 5$, March 2013, UK

implies that it is more probable to have a path, out of k paths, with a smaller travel time error at a particular time instant. Obviously, the average time error of TTP decreases more rapidly than the others. We also measure the computation time of TTP as shown in Figure 7(b). It increases with k because the number of k-combinations enumerated also increases. Figure 7(c) measures the frequency of real fastest path(s) appearing in the result path set, denoted as $SPfreq(P)$, for every method. When $k \geq 5$, our proposed solutions (TTP and TP) achieve over 95% frequency, whereas the competitors have at most 80% frequency.

Varying D: In this experiment, we set the number of time instants per day to one ($L = 1$) but increase the number of training days (D) from 15 to 60 in order to examine its effect on TTPand TP. The time error and computation time of TTP and TP are shown in Figure 8. In general, the time error of both TTP and TP drops with increasing D. However, the trade-off of TTP is the drastic increase in computation cost. The reason is that dominance loses selectivity when increasing the number of dimensions. An apparent advantage of TP over TTP is that its computation cost rises slightly with increasing m.

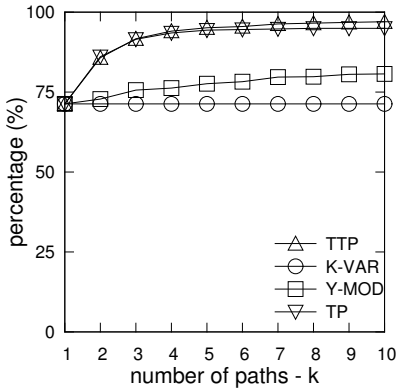
Varying L: Conversely, we fix the number of training days to 15 but vary the number of time instants per day (L) from 1 to 4. The same measurements are made and also presented in Figure 8. The time error of both TTP and TP rises when including more time instants. This may be because more time instants introduce higher traffic variability. Same as before, their computation costs increase proportionally to the number of dimensions but TP enjoys a much gentler rate of increase in computation time. To sum up, one time instant per day (e.g., 08:00 - 08:15) with 15 to 30 training days may be a feasible choice for TTP in UK road network, and TP could be considered as a close approximation of TTP by slightly sacrificing accuracy in travel time.

Varying distance from source: We evaluate the effect of the distance from source, keeping all other parameters as default. Figure 9 shows the results. The x-axis represents the query groups as described in Section 8.2; the distance threshold increases from the left to the right. The travel time errors of all four methods grows when the distance between query locations increases. However, the errors of Y-MOD and K-VAR rise drastically (about four times higher than TTP and TP). This implies that the paths returned by Y-MOD and K-VAR accumulate travel time delay more rapidly (about four times quicker) and are hence



(a) Average time error

(b) Computation time



(c) Frequency of real fastest path(s) appearing in the result path set

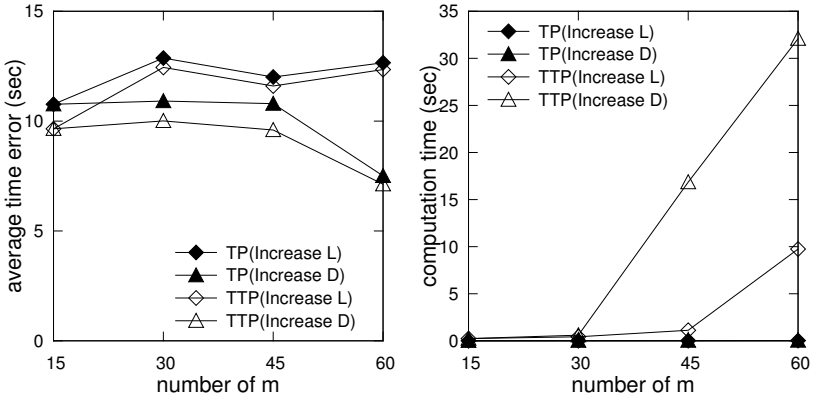
Fig. 7 Varying k, 08:00-08:15 in March 2013, UK

less resilient to live traffic. This experiment further justifies that TP could be a reasonable alternative to TTP (albeit heuristic) as it exhibits small errors.

8.4 Synthetic Traffic Data

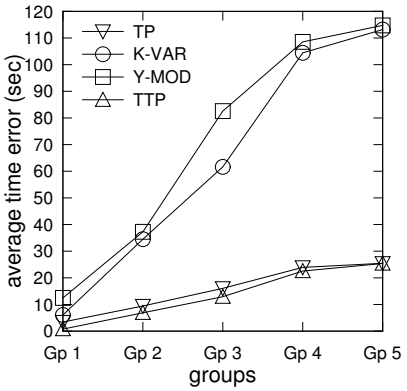
Since the exact solution is not scalable to large road networks, we developed two heuristics - TP and ATP. In this section, we evaluate mainly the error and efficiency of our proposed heuristics by varying the following parameters - (i) the percentage change of traffic time ($X\%$), (ii) the number of paths (k), (iii) the number of time instants (m), and (iv) increasing distance from sources. Note that uniformly random queries are used in (i) to (iii) while distance threshold queries are used in (iv).

Varying allowed time of ATP: Before showing the results of varying the mentioned parameters, we first show that the average time error of ATP drops initially and converges as allowed running time increases. This finding justifies the reason of proposing ATP. We only show the case of $k = 8, 9, 10$ because the computation times of TP in these cases are more



(a) Average time error (b) Computation time

Fig. 8 Varying m, fix $k = 5$, UK



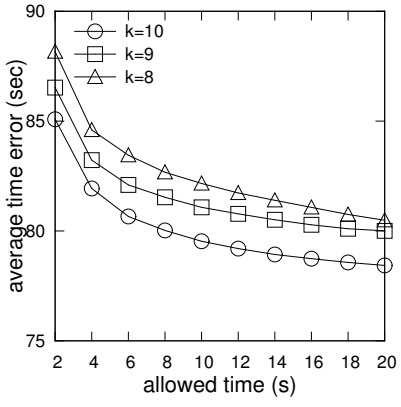
Average time error

Fig. 9 Varying distance from source, $N = 5$, fix other parameters as default, UK

than 30 seconds according to Figure 10. In the following discussions, we set the default time limit of ATP to 5 seconds and this is denoted by ATP-5 in the figures.

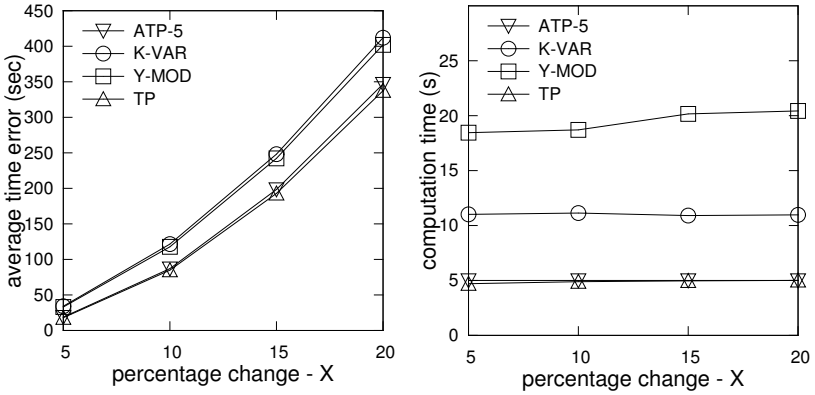
Varying X: In this experiment, we increase the traffic fluctuation and measure the changes of average time errors. The results are shown in Figure 11. The average time errors of the three methods rise when increasing the percentage change of travel time. Nevertheless, the errors of TP and ATP are still smaller than the others, implying that they are more resistant to the traffic fluctuation. We also show their computation times, which are insensitive to the traffic changes.

Varying k: Figure 12 shows the average time errors and computation times of all methods, and reveals important properties. The computation cost of K-VAR stabilizes after k becomes larger than 1 because it is controlled by a maximum number of iterations, which is fixed throughout the experiment and configured according to [21]. Its time error remains unchanged with k because, as we found out, it can retrieve only one path throughout its iterations. The computation time of Y-MOD increases linearly with k since it is a variant of



Average time error

Fig. 10 Varying allowed time of ATP, fix $X\% = 10$ and $m = 30$, COL

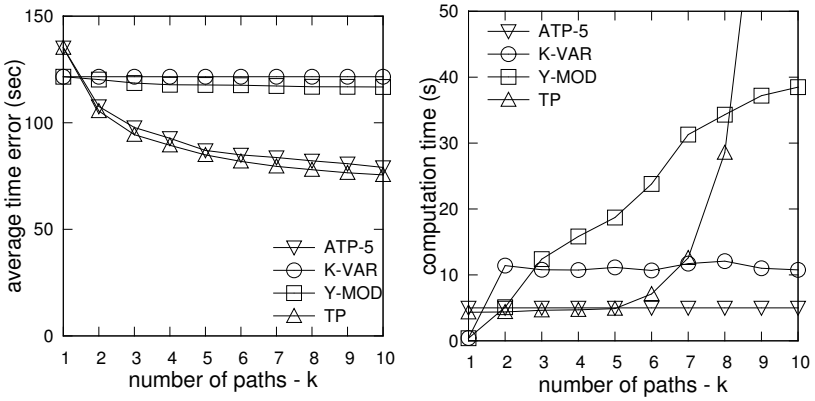


(a) Average time error (b) Computation time

Fig. 11 Varying $X\%$, fix $k = 5$ and $m = 30$, COL

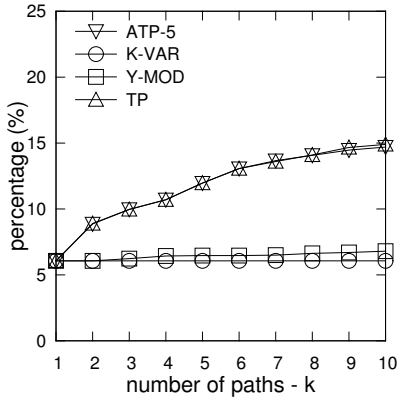
Yen's K shortest path algorithm [26] whose complexity is linearly dependent on k. However, its time error drops slightly with increasing k. This may be because of the inherent property of Yen's algorithm, which generates a set of K paths with a large portion of overlapping. As for TP, its computation time rises rapidly when k is larger than 5. The running times for $k = 9$ and 10 are 75 and 148 seconds respectively. The rise in computation cost is due to the exponential number of enumerated combinations. Despite this, the error of TP diminishes with k continuously and outperforms the others. ATP exhibits a similar diminishing trend to TP but it offers constant computation cost. To summarize, TP and ATP both strike a better trade-off between error and computation cost.

In addition to the average travel time error, we also report alternative error measures in Figure 12(c), Tables 9 and 10. First, we present the frequency of real fastest path(s) appearing in the result set, $SPfreq(P)$, in Figure 12(c). These frequencies are much lower than those on the UK network (cf. Figure 7(c)). This is probably attributed to the fact that



(a) Average time error

(b) Computation time



(c) Frequency of real fastest path(s) appearing in the result path set

Fig. 12 Varying k, fix $X\% = 10$ and $m = 30$, COL

the COL network is larger and contains much more feasible paths per query. Nevertheless, our methods TP and ATP-5 beat the competitors Y-MOD and K-VAR consistently.

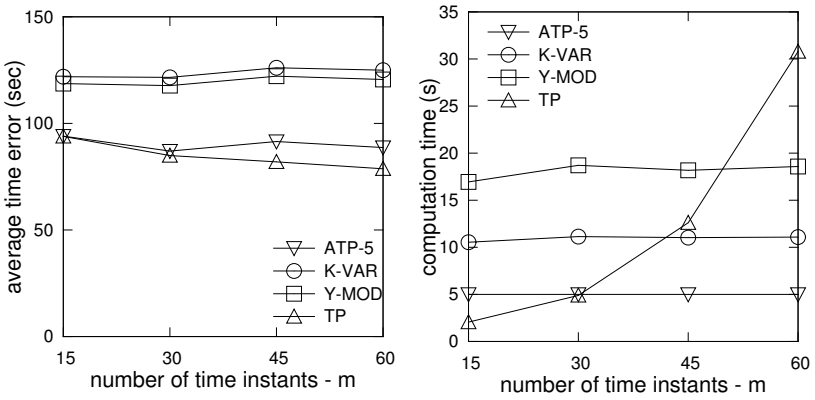
To investigate further, we focus on the case $k = 5$ and report in Table 9 the statistical error measures of the methods, namely ϵ^Q_{min}, $\epsilon^Q_{25\%}$, $\epsilon^Q_{50\%}$, $\epsilon^Q_{75\%}$, ϵ^Q_{max}. Regardless of the error measure used, our methods consistently produce better result paths than our competitors. We observe similar trends for other values of k, e.g., the case for $k = 10$ in Table 10. Our methods TP and ATP-5 yield less than 1 minute of error for the majority of cases, and the error decreases fast as k increases.

Table 9 Statistics of travel time errors (in seconds), $k = 5$, COL

Methods	ϵ^Q_{min}	$\epsilon^Q_{25\%}$	$\epsilon^Q_{50\%}$	$\epsilon^Q_{75\%}$	ϵ^Q_{max}
TTP	0	10	43	115	1,062
ATP-5	0	10	43	115	1,062
Y-MOD	0	19	70	168	1,344
K-VAR	0	21	73	172	1,434

Table 10 Statistics of travel time errors (in seconds), $k = 10$, COL

Methods	ϵ^Q_{min}	$\epsilon^Q_{25\%}$	$\epsilon^Q_{50\%}$	$\epsilon^Q_{75\%}$	ϵ^Q_{max}
TTP	0	7	36	104	857
ATP-5	0	7	37	107	937
Y-MOD	0	19	69	167	1,344
K-VAR	0	21	73	172	1,434



(a) Average time error (b) Computation time

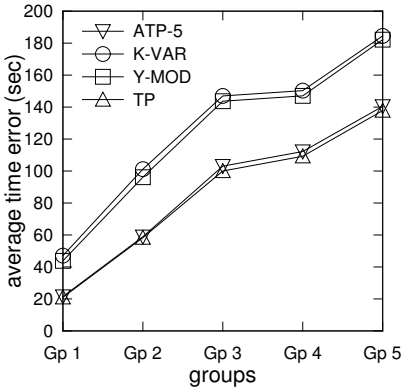
Fig. 13 Varying m, fix $X\% = 10$ and $k = 5$, COL

Varying m: Figure 13 shows the average time errors and computation times of the methods versus the number of time instants. The average time errors and computation times of Y-MOD and K-VAR are insensitive to m since they simply aggregate (i.e., average) the travel time of all time instants to a single travel time for candidate generation. When m increases, the average time error of TP drops slightly while its computation cost increases exponentially due to enumeration of combinations. As for ATP, its average time error decreases initially but increases afterwards. This is because it cannot further explore more paths upon reaching the time limit, i.e., 5 seconds.

Varying distance from source: Finally, we look into the change in travel time error with increasing distance from source. Figure 14 shows a generally growing trend of travel time error which matches the findings in UK road network. Our proposed heuristics still outperform the competitors and ATP achieves small errors, despite having a computation time limit.

8.5 Discussion

By comparing the experimental results of real and synthetic traffic data on two road networks, it is intriguing to induce that they are coherent and match each other. First of all, TTP and TP yield smaller average travel time errors than both competitors when we vary the number of paths returned (k), the number of time instants (m), and the distance between source and destination. Besides, the performance of our traffic-tolerant paths on real traffic data is apparently more preferable, proving that they are suitable and feasible in practice for answering route queries and assisting traffic analysis. Both sets of experiments also demon-



Average time error

Fig. 14 Varying distance from source, $N = 5$, fix other parameters as default, COL

strate that our proposed heuristics can closely approximate the exact solution, striking a balance between travel time error and computation time.

9 Conclusions

This paper proposes and studies a novel problem called the k traffic-tolerant paths problem (TTP) in road networks, which takes a source-destination pair and historic traffic information as input and returns k paths that minimize the aggregate historic travel time. Its applications include transportation analysis and efficient route-search services. We implement an exact enumeration algorithm for TTP. Since our TTP problem is NP-hard, we also devise two heuristics to solve it. Finally, the experiments show that we achieve much higher accuracy than existing approaches.

References

1. 9th DIMACS Implementation Challenge - Shortest Paths. http://www.dis.uniroma1.it/challenge9/
2. Caltrans Pems. http://pems.dot.ca.gov/
3. GB Road Traffic Counts. http://data.gov.uk/dataset/gb-road-traffic-counts/
4. Google Maps. http://maps.google.com/
5. Highways Agency Network Journey Time and Traffic Flow Data. http://data.gov.uk/dataset/dft-eng-srn-routes-journey-times/
6. TomTom - At the Heart of the Journey. http://www.tomtom.com/
7. Travel Time Reliability: Making It There On Time, All The Time. http://ops.fhwa.dot.gov/publications/tt_reliability/index.htm (2006)
8. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.F.: Hierarchical Hub Labelings for Shortest Paths. In: ESA, pp. 24–35 (2012)
9. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In Transit to Constant Time Shortest-path Queries in Road Networks. In: ALENEX (2007)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition, 3rd edn. The MIT Press (2009)
11. Demiryurek, U., Kashani, F.B., Shahabi, C., Ranganathan, A.: Online Computation of Fastest Path in Time-dependent Spatial Networks. In: SSTD, pp. 92–111 (2011)

12. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In: Proceedings of the 7th International Conference on Experimental Algorithms, WEA'08, pp. 319–333 (2008)
13. 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, pp. 794–805 (2007)
14. González, M.C., Hidalgo, C.A., Barabási, A.L.: Understanding individual human mobility patterns. Nature **453**(7196), 779–782 (2008)
15. Gutman, R.: Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In: ALENEX, pp. 100–111 (2004)
16. Hua, M., Pei, J.: Probabilistic Path Queries in Road Networks: Traffic Uncertainty Aware Path Selection. In: EDBT, pp. 347–358 (2010)
17. Kanoulas, E., Du, Y., Xia, T., Zhang, D.: Finding Fastest Paths on a Road Network with Speed Patterns. In: ICDE, pp. 10–10 (2006)
18. Kriegel, H.P., Renz, M., Schubert, M.: Route Skyline Queries: A Multi-Preference Path Planning Approach. In: ICDE, pp. 261–272 (2010)
19. Li, P.H., Yiu, M.L., Mouratidis, K.: Historical Traffic-tolerant Paths in Road Networks. In: ACM GIS, to appear (2014)
20. Lomax, T., Schrank, D., Turner, S., Margiotta, R.: Selecting travel reliability measures. Texas Transportation Institute monograph (May 2003) (2003)
21. Malviya, N., Madden, S., Bhattacharya, A.: A Continuous Query System for Dynamic Route Planning. In: ICDE, pp. 792–803 (2011)
22. Sanders, P., Schultes, D.: Highway Hierarchies Hasten Exact Shortest Path Queries. In: ESA, pp. 568–579 (2005)
23. Sankaranarayanan, J., Samet, H.: Query Processing Using Distance Oracles for Spatial Networks. IEEE Trans. Knowl. Data Eng. **22**(8), 1158–1175 (2010)
24. Sankaranarayanan, J., Samet, H., Alborzi, H.: Path Oracles for Spatial Networks. PVLDB **2**(1), 1210–1221 (2009)
25. Song, C., Qu, Z., Blumm, N., Barabási, A.L.: Limits of predictability in human mobility. Science **327**(5968), 1018–1021 (2010)
26. Yen, J.Y.: Finding the K Shortest Loopless Paths in a Network. Management Science **17**(11), 712–716 (1971)
27. Zhu, A.D., Ma, H., Xiao, X., Luo, S., Tang, Y., Zhou, S.: Shortest Path and Distance Queries on Road Networks: Towards Bridging Theory and Practice. In: SIGMOD, pp. 857–868 (2013)