Congestion-mitigating Spatiotemporal Routing in Road Networks

Libin Wang[†], Raymond Chi-Wing Wong[†], Christian S. Jensen[‡]
[†]The Hong Kong University of Science and Technology, Hong Kong SAR, China
[‡]Aalborg University, Aalborg, Denmark
{lwangct, raywong}@cse.ust.hk, csj@cs.aau.dk

***Abstract*—Vehicular traffic congestion is a recurring and widespread societal phenomenon. Since drivers increasingly rely on routing services, we consider how to enhance such services to provide routes that mitigate congestion. Specifically, we propose the Congestion-mitigating Spatiotemporal Routing (CSR) problem that considers the congestion caused by vehicles following the routes provided. This problem is challenging because vehicles that follow recommended routes appear on different road segments at different times. We propose two solutions, Spatiotemporal Oblivious Routing (SOR) and Spatiotemporal Routing with History (SRH), which return routes based on the current and anticipated future traffic statuses, respectively, while offering theoretical guarantees. We also propose an update procedure for handling traffic dynamics. Extensive evaluations on real data provide insight into the properties of the solutions, indicating that SRH can reduce the number of vehicles on the most congested road segments by nearly 33% and can process a query in less than 10 ms.**

I. Introduction

The lives of people in cities are frequently affected by road-network congestion. For instance, commuters are often affected by the daily morning and afternoon rush hour that may cause substantial delays. In extreme cases, drivers were stuck on an expressway for days and moved just two miles per day [1]. Apart from inconveniencing drivers, congestion incurs financial and environmental costs at a societal level [2]. Therefore, congestion reduction is highly desirable.

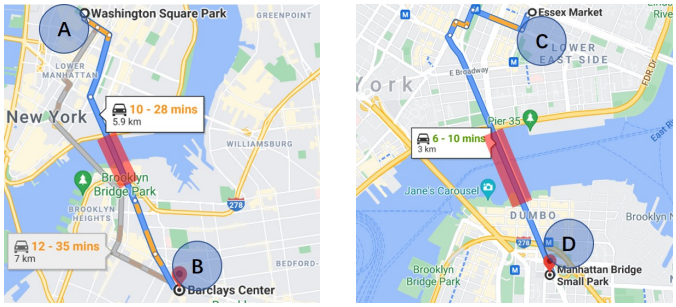
The proliferation of smartphones and personal navigation devices has spawned many routing services, including online mapping apps (e.g., Google Maps and Baidu Maps), taxi-calling platforms (e.g., Uber and Lyft), and food delivery services (e.g., Uber Eats and Deliveroo). Drivers increasingly rely on such services for finding the fastest routes to their destinations. However, studies show that vehicles following such fastest routes that take into account only the needs of a single driver may aggravate congestion [3]–[5]. Since drivers are relying increasingly on routing services, there is now an opportunity to reduce congestion by enabling routing services to mitigate congestion by sometimes returning slightly slower routes with acceptable detours.

Providing routing that reduces congestion is non-trivial in real traffic. First, services receive queries continuously and must return results immediately. From a global perspective, previously returned routes may have adverse effects on the route to return for a current query. Put differently, we cannot take into account future queries to make optimal routing decisions for a query. Second, recommended routes may conflict and increase congestion under certain circumstances. Thus, when several vehicles utilize the same road segments at the same time, they increase the congestion of that segment. We therefore need to keep track of how computed routes contribute to the congestion of road segments over time.

A line of research focuses on finding shortest or fastest routes efficiently [6]–[9], but their deterministic route choices for similar queries disregard congestion and may thus increase congestion. Other studies consider alternative routes [10]–[16]. However, their choices of "good" routes are subjective and do not aim to mitigate congestion. Recent transportation studies propose a variety of heuristics (e.g., the entropy method [17]) to reduce congestion [17]–[21]. Nevertheless, these neither offer theoretical guarantees nor scale large road networks and frequent online routing queries. Recent studies also exit that process routing queries in batches to minimize sums of travel times [22], [23], but such batching is at odds with the need for instant query results.

Motivated by the above challenges, we formulate the new *Congestion-mitigating Spatiotemporal Routing (CSR)* problem. This involves keeping track of the time-varying congestion degrees of road segments and attempting to reduce the congestion degree of the most congested road segment, which is a likely traffic bottleneck. To solve the problem, we propose two algorithms that aim to enable instant responses to online routing queries that mitigate congestion. First, *Spatiotemporal Oblivious Routing (SOR)* recommends routes by considering the existing traffic status and the time-varying congestion degrees of road segments. Second, based on SOR, *Spatiotemporal Routing with History (SRH)* utilizes historical information to anticipate future congested road segments to further mitigate congestion. Moreover, since the travel times of road segments vary over time, we design procedures for the two algorithms to handle traffic updates efficiently. To give some intuition, we present the following toy example with the fastest route as the baseline.

Example 1: In New York City, a service at some point receives 20 queries from 20 cars with the same source and destination pair, A and B, as shown in Figure 1a. The fastest route traverses the bridge in red, which makes it the most congested road segment with 20 cars. Algorithm SOR that considers the existing traffic status returns the gray route that



(a) First type of queries (b) Second type of queries

Fig. 1: An example of routing queries

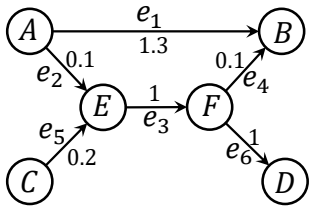


Fig. 2: The road network of Example 1

traverses the neighboring bridge to 10 cars, meaning that the most congested segment has 10 cars. Next, assume that we also receive 20 queries in short succession, all from C to D, as shown in Figure 1b. They can only use the route traversing the red bridge since traversing the neighboring bridge takes too long. With SOR, this will result in 30 cars on the most congested segment. Algorithm SRH uses historical data and is able to foresee this future congested segment. Thus, it can return the gray route to the first 20 queries from A to B and then use the red bridge for the last 20 queries.

We summarize our contributions as follows.

- We formulate the congestion reduction problem called the Congestion-mitigating Spatiotemporal Routing problem (CSR). We prove that it is NP-hard in an offline setting.
- We propose two algorithms, SOR and SRH, that leverage the existing and future traffic statuses, respectively. They enable a congestion degree on the most congested road segment of at most $\mathcal{O}(\ln n)$ and $\mathcal{O}(\ln |\mathcal{C}|)$ the optimal value, where n is the number of vertices and C is a set of candidate congested road segments, which is small and contains on the order of $10k$ segments in practice.
- We report on experiments on real-world data that provide insight into the performance of the propped algorithms, showing that they can reduce the load of the most congested road segment by 33% over a baseline and can process queries in less than 10 ms.

II. Problem Statement

A. Problem Definitions

Definition 1 (Dynamic Road Network): A road network $G(V, E, W)$ is a connected directed graph, where V and E are the node and edge sets, respectively, with $n = |V|$ and $m = |E|$, and $W : E \to R^+$ is an updatable function that assigns a travel time w_e to each edge e.

Similar to [9], [24], we consider a dynamic network setting where edge weights can be updated. We initially assume static weights and then extend the coverage to dynamic weights in Section IV.

Definition 2 (Path): A path p is a finite sequence of edges $p = \langle e_1, e_2, \ldots, e_k \rangle$ where the ending vertex of e_i is the starting vertex of e_{i+1}, $1 \leq i < k$. Its travel time $\gamma_p = \sum_{i=1}^{k} w_{e_i}$. Let s_p and d_p be the starting vertex of e_1 and the ending vertex of e_k, respectively.

Definition 3 (Routing Query): Each routing query $q = (t_q, s_q, d_q)$ is issued by a vehicle with its departure time $t_q \in \mathbb{R}_{\geq 0}$, its source s_q, and its destination d_q.

Definition 4 (Fastest Path): Given a routing query $q = (t_q, s_q, d_q)$, the fastest path $f^*(q)$ is defined to be the one with the shortest travel time among all paths from s_q to d_q.

Example 2: In Figure 2, we abstract a road network from Example 1. There are 6 nodes and 6 edges, with edge weights shown next to them. For a query $q = (0, A, B)$, the fastest path $f^*(q) = (e_2, e_3, e_4)$ since its travel time is smaller than that of path (e_1).

Definition 5 (Detour Constraint): A routing query $q = (t_q, s_q, d_q)$ is answered by a path p with travel time γ_p no larger than $(1+a)$ times that of the shortest travel time $\gamma_{f^*(q)}$:

$$\gamma_p \leq (1 + a)\gamma_{f^*(q)}, \tag{1}$$

where $a \geq 0$ is a detour factor used to control the detour cost. Let $f(q)$ to denote any path satisfying the detour constraint.

Example 3: In Example 2, suppose that we first receive two routing queries $q_1 = q_2 = (0, A, B)$ and the detour factor $a = 0.1$. We may return the path $f(q) = (e_1)$ with travel time 1.3 since $(1 + a)\gamma_{f^*(q)} = (1 + 0.1) \cdot 1.2 = 1.32$.

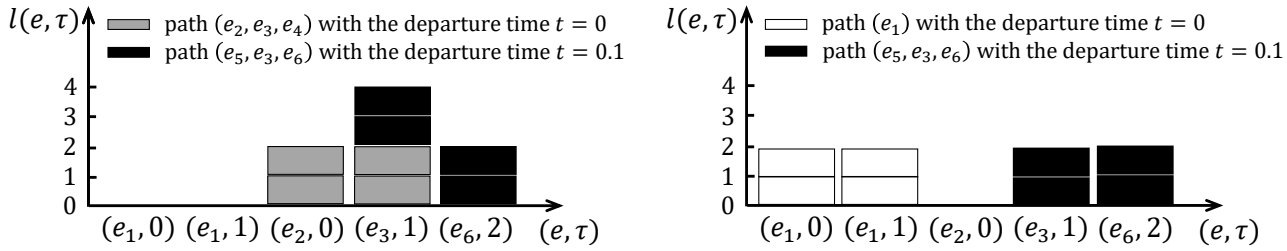
When vehicles move along the recommended paths, they appear on different edges at different times, which makes the *traffic status* change over time. To examine the traffic status efficiently, we only do so at so-called time steps.

Definition 6 (Time Step): The timeline is first partitioned into equal-length intervals (e.g., one minute long). Then, the traffic status is updated at the end of each interval, i.e., at the times $t = 0, 1, \ldots$ (where t are integers, not real numbers), called time steps, denoted by $\tau \in \mathbb{N}$. We ignore fluctuations in the traffic status within the duration of a time step.

In reality, one minute per step is sufficient since this is also the frequency that Baidu Maps uses to update its traffic status [25]. Then, $t = 0.1$ is 6 seconds from $t = 0$. Let T be the set of all time steps of interest. To capture the traffic status, we define the notion of *edge load* across time steps.

Definition 7 (Edge Load): There is an edge load function $l : E \times T \to \mathbb{N}$ such that the edge load $l(e, \tau)$ measures the traffic status of the *edge-step pair* (e, τ) for each edge $e \in E$ and step $\tau \in T$.

The solution we provide can be extended easily to supporting intervals with varying lengths. We only need to measure each load $l(e, \tau)$ at the actual time corresponding to step τ.



(a) The edge loads of the fastest path oracle

(b) The edge loads of the optimal solution

Fig. 3: The edge loads of two algorithms for the queries in Example 1

We finally formulate the congestion-mitigating routing problem as follows.

Definition 8 (Congestion-mitigating Spatiotemporal Routing (CSR)): Given a network G and a detour factor a, we receive online routing queries $q = (t_q, s_q, d_q)$ one by one and have to process them right away. We want to build an oracle f that given a query q returns a path $f(q)$ that satisfies the detour constraint (Equation 1) and minimizes the maximum edge load incurred by the paths returned for all queries received so far:

$$\min_{f} \max_{e \in E, \tau \in T} l(e, \tau). \tag{2}$$

B. Edge Load

We focus on a particular edge load type, called the volume-to-capacity (v/c) ratio, which is used widely in the transportation area to quantify congestion [26]–[28]. We next define it formally in the context of our problem.

Definition 9 (Location Indicator): Given a path $p = \langle e_1, \ldots, e_k \rangle$ with departure time t, the location indicator $\mathbb{1}_{p,t}(e_i, \tau)$ is defined as 1 if and only if 1) e_i is in p and 2) the vehicle following p and departing at t is on e_i at step τ, i.e., $t + \gamma_{(e_1,\ldots,e_{i-1})} \leq \tau < t + \gamma_{(e_1,\ldots,e_i)}$ for $i > 1$ and $t \leq \tau < t + \gamma_{(e_1)}$ for $i = 1$.

When the departure time of a query is clear from the context, we use $\mathbb{1}_p(e, \tau)$. We simply say that the path p traverses the edge-step pair (e, τ) if $\mathbb{1}_p(e, \tau) = 1$.

Example 4: Consider a path $p = (e_5, e_3, e_6)$ with departure time $t = 0.1$ in Figure 2. The location indicators for edge-step pairs $(e_3, 1)$ and $(e_6, 2)$ are 1. $\mathbb{1}_p(e_3, 1) = 1$ because $0.1 + w_{e_5} = 0.3 \leq 1 < 0.1 + w_{e_5} + w_{e_3} = 1.3$, and $\mathbb{1}_p(e_6, 2) = 1$ because $0.1 + w_{e_5} + w_{e_3} = 1.3 \leq 2 < 0.1 + w_{e_5} + w_{e_3} + w_{e_6} = 2.3$. For any step τ, $\mathbb{1}_p(e_5, \tau) = 0$ because no step $\tau \in \mathbb{N}$ can be in the interval $[t, t + w_{e_1}) = [0.1, 0.3)$. Intuitively, e_5 is so short that the vehicle that traverses it does not show at any step when we examine the traffic status.

Definition 10 (Edge Volume): Given a set P of paths with corresponding departure times, we define the edge volume $v(e, \tau)$ as the number of paths (or vehicles) traversing the edge-step pair (e, τ), i.e., $v(e, \tau) = \sum_{p \in P} \mathbb{1}_p(e, \tau)$.

Definition 11 (Edge Capacity): The edge capacity $c(e)$ is the total number of vehicles that can traverse the edge e at a given speed without delay. We assume that an oracle has access to this function, which can be determined by the edge's distance, type, and number of lanes [29].

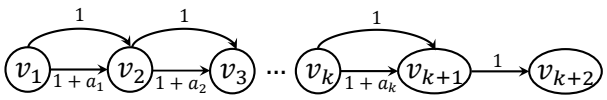


Fig. 4: Graph G in the NP-hardness proof

Definition 12 (Edge Load): We define the edge load $l_P(e, \tau)$ incurred by P as the edge volume over the edge capacity, i.e., $l_P(e, \tau) = v(e, \tau)/c(e)$.

Example 5: We use queries similar to those in Example 1 to illustrate the edge load and the CSR problem. Let $c(e) = 1$ for all edges for ease of illustration. We use an oracle that returns the fastest path $f^*(q)$ as a baseline. Suppose that we first receive queries $q_1 = q_2 = (0, A, B)$. The fastest path for them is (e_2, e_3, e_4) with departure time $t = 0$. The loads that they incur are shown as the grey blocks in Figure 3a, where each block indicates that the corresponding location indicator is 1. Next, we receive queries $q_3 = q_4 = (0.1, C, D)$. We can only return the path (e_5, e_3, e_6) with departure time $t = 0.1$, shown as the black blocks in Figure 3a. The maximum edge load for the fastest path oracle is thus $l(e_3, 1) = 4$. However, an optimal oracle returns path (e_1) for the first two queries, incurring the edge loads shown by the white blocks in Figure 3b. The optimal maximum edge load is thus 2.

C. Hardness

CSR is an online problem in the sense that when answering a query, we know nothing about future queries, which makes any algorithms suboptimal compared to one that has full information. We can obtain the *hypothetically* optimal solution by first collecting all the queries up to now without handling them and solving the problem in hindsight. However, even if we know all the queries beforehand, also known as the offline version of this problem, the problem is NP-hard.

Theorem 1: Offline CSR is NP-hard.

Proof: We use the reduction from *the subset sum problem* [30], which is NP-complete, to the decision version of CSR. The subset sum problem considers the numbers $b, a_1, \ldots, a_k \in \mathbb{Z}^+$ and asks whether there exist $x_1, \ldots, x_k \in \{0, 1\}$ such that $b = x_1 a_1 + \cdots + x_k a_k$. The decision version of CSR asks whether the maximum edge load (Equation 2) could be at most some integer $L > 0$.

Given an instance of the subset sum problem, we construct the graph G in Figure 4. There are exactly 2^k paths from source v_1 to destination v_{k+1}; each top edge represents $x_i = 0$,

and each bottom one represents $x_i = 1$. We receive a query $q_1 = (0, v_1, v_{k+2})$ and L queries (τ, v_{k+1}, v_{k+2}) for each step $\tau \in \mathbb{N}$ except for step $\tau = b + k$. For query q_1, we observe that if we route a path with its travel time exactly equal to $b+k$ (i.e., a yes answer to the subset sum problem), there are no L queries (τ, v_{k+1}, v_{k+2}) for $\tau = b + k$, which makes the maximum load equal to L (i.e., a yes answer to CSR). Paths with travel times other than $b + k$ result in a maximum load equal to $L + 1$. Therefore, the subset sum problem has a yes answer if and only if the corresponding decision version of CSR has a yes answer. ∎

Let Q be the set of queries received so far. Let $ALG(Q)$ and $OPT(Q)$ be the objective values (Equation 2) of an algorithm and the optimal solution on Q. We would like to design an online algorithm ALG such that for any input queries received so far: $ALG(Q) \leq c \cdot OPT(Q)$. This is also called a c-competitive algorithm, and c is called the *competitive ratio*.

III. Routing Algorithms

A. Oblivious Online Routing

Given a routing query q, we need to return one path from the set $\mathcal{P}(q) = \{p | s_p = s_q, d_p = d_q, \gamma_p \leq (1 + a)\gamma_{f^*(q)}\}$ that contains all paths that satisfy the detour constraint (Equation 1). To select $p \in \mathcal{P}(q)$ that increases congestion the least, we first define a metric function to evaluate each path and then select the one that minimizes the function. Specifically, since different paths traverse different edges-step pairs (e, τ), we assign a variable $x(e, \tau)$ for each edge-step pair (e, τ) to indicate the current congestion degree for (e, τ). For each path $p \in \mathcal{P}(q)$, we use the sum of variables $x(e, \tau)$ w.r.t. those edge-step pairs that p traverses (i.e., $\mathbb{1}_p(e, \tau) = 1$) as the metric value.

A heuristic idea may directly set $x(e, \tau)$ as the current edge load $l(e, \tau)$ incurred by existing paths that we return to all the queries so far. However, it cannot handle the case where the edge loads caused by existing paths are unevenly distributed, as exemplified next.

Example 6: Suppose that the current edge loads $l(e_1, 0) = 4$, $l(e_1, 1) = 4$, $l(e_2, 0) = 1$, and $l(e_3, 1) = 7$ and that $c(e) = 1$ for all edges for ease of illustration. Consider a query $q = (0, A, B)$. The path (e_1) traverses $(e_1, 0)$ and $(e_1, 1)$ and has metric value $l(e_1, 0) + l(e_1, 1) = 8$, and the other path (e_2, e_3, e_4) traverses $(e_2, 0)$ and $(e_3, 1)$ with the same metric value $l(e_2, 0) + l(e_3, 1) = 8$, which indicates that we can select either of the two paths. However, we actually prefer the path (e_1) because the maximum load after we select (e_1) is $l(e_3, 1) = 7$ since $l(e_1, 0) = l(e_1, 1) = 5$, and $l(e_3, 1)$ is still the largest value, whereas after we select (e_2, e_3, e_4), the maximum load $l(e_3, 1) = 7 + 1 = 8$, which is larger.

Apart from the above weakness, directly using the edge load builds no connection with the optimal solution. Instead, SOR uses an exponential growth function $\alpha\beta^{v(e,\tau)}$ for $x(e, \tau)$ that places $v(e, \tau)$ in the exponent. In Example 6, we select the path (e_1) because $\beta^4 + \beta^4 < \beta^7 + \beta$ (where the volumes are used as the exponents) for any $\beta > 0$ and $\beta \neq 1$ by the inequality of arithmetic and geometric means. It also considers

Algorithm 1: Spatiotemporal Oblivious Routing (SOR)

1 $x(e, \tau) \leftarrow \frac{1}{2Umc(e)}$ and $v(e, \tau) \leftarrow 0$ for each (e, τ)
  $\lambda \leftarrow \min_e \frac{1}{c(e)}$
2 **foreach** *new query* q **do**
3 $\mathcal{P}(q) \leftarrow \{p | s_p = s_q, d_p = d_q, \gamma_p \leq (1 + a)\gamma_{f^*(q)}\}$
4 $p \leftarrow \arg\min_{p \in \mathcal{P}(q)} \sum_{e,\tau} \mathbb{1}_p(e, \tau)x(e, \tau)$
5 **while** $\sum_{e,\tau} \mathbb{1}_p(e, \tau)x(e, \tau) > \lambda$ *or there exists one* $x(e, \tau) > \frac{\exp(\frac{1}{2})}{c(e)}$ **do**
6 $\lambda \leftarrow 2\lambda$
7 $x(e, \tau) \leftarrow \frac{(1+\frac{1}{2\lambda c(e)})^{v(e,\tau)}}{2Umc(e)}$ for all e, τ
8 $p \leftarrow \arg\min_{p \in \mathcal{P}(q)} \sum_{e,\tau} \mathbb{1}_p(e, \tau)x(e, \tau)$
9 use the path p for this query q, i.e., $f(q) = p$
10 **foreach** e, τ *such that* $\mathbb{1}_{f(q)}(e, \tau) = 1$ **do**
11 $v(e, \tau) \leftarrow v(e, \tau) + 1$
12 $x(e, \tau) \leftarrow (1 + \frac{1}{2\lambda c(e)})x(e, \tau)$

an estimate of the optimal solution in the function to assess its performance. Specifically, we define

$$x(e, \tau) = \frac{(1 + \frac{1}{2\lambda c(e)})^{v(e,\tau)}}{2Umc(e)}, \tag{3}$$

where $m = |E|$, $U = \lceil \max_p \gamma_p \rceil$, and $\lambda > 0$ is an estimate of the optimal solution, which is initially set to $\min_e \frac{1}{c(e)}$ (corresponding to the minimum load for a path) and increases under some conditions where the optimal maximum load (denoted by OPT) increases. We can view the variable $x(e, \tau)$ as $\alpha\beta^{v(e,\tau)}$ where $\alpha = \frac{1}{2Umc(e)}$ and $\beta = 1 + \frac{1}{2\lambda c(e)}$. This overcomes the first weakness stated above. To make each $v(e, \tau)$ (i.e., the exponent) not very large, our algorithm ensures that $x(e, \tau)$ can reach at most a constant upper bound at any time. If $x(e, \tau)$ exceeds the upper bound, this indicates that the estimate λ of the optimum increases (proved later), and we can make $x(e, \tau)$ smaller than the upper bound by increasing λ.

Algorithm 1 summarizes the procedure. In Line 1, we initialize each variable $x(e, \tau)$ to $\frac{1}{2Umc(e)}$ (since $v(e, \tau)$ is 0 in Equation 3) and the initial estimate of the optimal solution λ to $\min_e \frac{1}{c(e)}$ corresponding to the optimal value for a path. For each new query q, we find the path set $\mathcal{P}(q)$ and the path with the minimum sum of the variables in Lines 3–4. In Lines 9–12, after returning path $f(q)$ for the new query, we update the variables regarding the edge-step pairs that path $f(q)$ traverses (i.e., $\mathbb{1}_{f(q)}(e, \tau) = 1$) by multiplying them with $(1 + \frac{1}{2\lambda c(e)})$ in Line 12 (since their edge volumes $v(e, \tau)$ are increased by 1 in Line 11). In Lines 5–8, we mainly update the estimate of the optimal solution λ. In Line 5, the two conditions indicate that the estimate may be smaller than the optimal maximum load OPT (proved later). Intuitively, the two conditions mean that either the sum of variables or one single variable is very large. We double the estimate of the optimal solution λ in

Line 6. Note that doubling λ makes all the variables $x(e,\tau)$ smaller according to Equation 3. We also reset all the variables in Line 7 and find the path under the new λ again in Line 8.

Note that the time-consuming part lies in Lines 3–4 of finding paths. We implement it by using a DFS search with pruning rules [31] with its time complexity $\mathcal{O}(m \ln n)$.

Example 7: Back to Example 5, we set each variable $x(e,\tau) = \frac{1}{2Umc(e)} = \frac{1}{36}$ since there are 6 edges and $U = 3$ (where the longest travel time of (e_5, e_3, e_6) is 2.2 and $U = \lceil 2.2 \rceil = 3$) and $\lambda = 1$. For the first query $q_1 = (0, A, B)$, the sum of variables for the path (e_1) is $x(e_1, 0) + x(e_1, 1) = \frac{1}{18}$, since only $\mathbb{1}_p(e_1, 0)$ and $\mathbb{1}_p(e_1, 1)$ are 1. Similarly, the sum for the path (e_2, e_3, e_4) is $x(e_2, 0) + x(e_3, 1) = \frac{1}{18}$. Algorithm 1 may arbitrarily select the path (e_1). The two variables $x(e_1, 0)$ and $x(e_1, 1)$ are all updated as $(1+\frac{1}{2\lambda c(e)})\frac{1}{36} = \frac{1}{24}$. For the query $q_2 = (0, A, B)$, Algorithm 1 will definitely select the path (e_2, e_3, e_4), since its sum of the variables $x(e_2, 0) + x(e_3, 1) = \frac{1}{18}$ is smaller than that of the path (e_1), which is $x(e_1, 0) + x(e_1, 1) = \frac{1}{12}$. Next, the two variables $x(e_2, 0)$ and $x(e_3, 1)$ are updated as $\frac{1}{24}$ similar to the previous update. For the next two queries $q_3 = q_4 = (0.1, C, D)$, we can only use (e_5, e_3, e_6). The maximum edge load achieved by Algorithm 1 is 3, since $l(e_3, 1) = \mathbb{1}_{(e_2,e_3,e_4)}(e_3, 1) + 2\mathbb{1}_{(e_5,e_3,e_6)}(e_3, 1) = 3$ and the edge loads of other edge-step pairs are smaller.

In practice, there could have been many existing external vehicles on the roads that would follow their own routes unknown to us. We may be only aware of the current number of vehicles on each road segment. If we want to take the current traffic status into account, we could reflect those existing vehicles in each variable $x(e,\tau)$ for all edges and only the current step. Specifically, suppose that there are currently k external vehicles on edge e at step τ. Since the exponent $v(e,\tau)$ in the variable $x(e,\tau)$ (Equation 3) should represent the current edge volume or the number of vehicles, we additionally multiply variable $x(e,\tau)$ by $(1+\frac{1}{2\lambda c(e)})^k$ to make the exponent consistent. Note that we have to update $x(e,\tau)$ in each step according to the current traffic status since we do not have their complete routes.

Since λ doubles whenever $x(e,\tau) > \frac{exp(\frac{1}{2})}{c(e)}$ in Line 5, we actually ensure this way that each variable representing the corresponding traffic status is always smaller than a constant.

Lemma 1: The edge load for each edge-step pair (e,τ) incurred by all the paths is $\mathcal{O}(\lambda \ln n)$, in terms of the final estimate of the optimal solution λ.

Proof: We initialize each variable $x(e,\tau)$ as $\frac{1}{2Umc(e)}$ and increase it to at most $(1 + \frac{1}{2\lambda c(e)})\frac{\exp(\frac{1}{2})}{c(e)} \leq \frac{3\exp(\frac{1}{2})}{2c(e)}$ (since $\lambda c(e) \geq 1$). Using Equation 3 for $x(e,\tau)$, we have $\frac{3\exp(\frac{1}{2})}{2} \geq \frac{(1+\frac{1}{2\lambda c(e)})^{v(e,\tau)}}{2Um} \geq \frac{(1+\frac{1}{2})^{\frac{v(e,\tau)}{\lambda c(e)}}}{2Um}$, where the second inequality is because $(1 + \frac{1}{2b}) \geq (1 + \frac{1}{2})^{\frac{1}{b}}$ for $b \geq 1$ and $\lambda c(e) \geq 1$. After taking the natural logarithm on both two sides and replacing $\frac{v(e,\tau)}{c(e)}$ by $l(e,\tau)$, we obtain $l(e,\tau) = \mathcal{O}(\lambda \ln n)$. ∎

Suppose that the last departure time of all the paths so far is t' and let $\tau' = \lfloor t' \rfloor$. We next show that $\lambda/2 < OPT$, which further makes the $\mathcal{O}(\ln n)$ competitive ratio, since any load $l(e,\tau) = \mathcal{O}(\ln n) \cdot OPT$. Let $OPT_{[0,\tau']}$ and $OPT_{[\tau'+1,\tau'+U]}$ be the maximum load of the optimal solution from step 0 to τ' and step $\tau'+1$ to $\tau'+U$, respectively. We first prove the statement on $OPT_{[\tau'+1,\tau'+U]}$).

Lemma 2: If $\sum_{e,\tau} \mathbb{1}_p(e,\tau)x(e,\tau) > \lambda$ or $x(e,\tau) > \exp(\frac{1}{2})$ for one edge-step pair (e,τ), the current estimate λ is smaller than $OPT_{[\tau'+1,\tau'+U]}$.

Proof: The idea of using the primal and dual programs is based on [32]. We construct the following linear program where $y(p)$ is the variable indicating whether we select the path p in the solution. The first set of constraints states that we could only select one path $p \in \mathcal{P}(q)$ for a query q. The second set of constraints state that for each pair (e,τ), its load should be smaller than λ after we move λ in the RHS. If $\lambda \geq OPT_{[\tau'+1,\tau'+U]}$, there exists a feasible solution (i.e., OPT) with its objective value equal to $|Q|$, where Q is the set of queries up to t'. If its optimal objective value is smaller than $|Q|$, we know that $\lambda < OPT_{[\tau'+1,\tau'+U]}$.

$$\begin{aligned} \max \quad & \sum_{q\in Q}\sum_{p\in\mathcal{P}(q)} y(p) \\ s.t. \quad & \textstyle\sum_{p\in\mathcal{P}(q)} y(p) \leq 1, \quad \forall q \in Q \\ & \textstyle\sum_{p:\mathbb{1}_{p,t}(e,\tau)=1} \frac{y(p)}{\lambda c(e)} \leq 1, \quad \forall e \in E, \tau'+1 \leq \tau \leq \tau'+U. \end{aligned}$$

We can find its dual program as follows. If the dual objective value of any feasible solution is smaller than $|Q|$, the primal one is also smaller than $|Q|$ and $OPT_{[\tau'+1,\tau'+U]} > \lambda$.

$$\begin{aligned} \min \quad & \sum_{e\in E}\sum_{\tau=\tau'+1}^{\tau'+U} c(e)x(e,\tau) + \sum_{q\in Q} z(q) \\ s.t. \quad & \textstyle\sum_{e,\tau:\mathbb{1}_{p,t}(e,\tau)=1} x(e.\tau)/\lambda + z(q) \geq 1, \quad \forall q \in Q, p \in \mathcal{P}(q). \end{aligned}$$

Let $\theta = \min_{p\in\mathcal{P}(q)} \sum_{e,\tau} \mathbb{1}_p(e,\tau)x(e,\tau)$ and $\Gamma = \theta/\lambda$. We construct a feasible dual solution as follows. For each $q \in Q$, we set the variable $x(e,\tau)$ as in Equation 3 and $z(q) = 1 - \Gamma$ so that the constraint is met with equality. Each time we route a path, the dual value increases at most $\Gamma/2 + 1 - \Gamma = 1 - \Gamma/2$. For the last query q, if we find $\theta > \lambda$ for some path p, the dual solution has already been feasible since all the constraints for potential paths related to this query are satisfied. We have a feasible dual solution with its objective value at most $\frac{1}{2} + |Q| - 1 < |Q|$ (where we use 1 as an upper bound for each increase of $1 - \Gamma/2$ and the initial dual value is $\frac{Um}{2Um} = \frac{1}{2}$), which further indicates $OPT_{[\tau'+1,\tau'+U]} > \lambda$.

When one $x(e,\tau)c(e) > \exp(\frac{1}{2})$, we can only consider those iterations where the corresponding $x(e,\tau)c(e)$ is increased from 1 to $\exp(\frac{1}{2})$ and use 1 as the upper bound for other iterations. There are at least $\lambda c(e)$ iterations since $(1 + \frac{1}{2\lambda c(e)})^{\lambda c(e)} \leq \exp(\frac{1}{2})$. When $x(e,\tau)c(e) \geq 1$, we also have $\Gamma \geq \frac{x(e,\tau)}{\lambda} \geq \frac{1}{\lambda c(e)}$. We set $z(q) = 1$ for the last request to make a feasible dual solution with its objective value strictly smaller than $\frac{1}{2} + |Q| - 1 - \lambda c(e) + \lambda c(e)(1 - \frac{1}{2\lambda c(e)}) + 1 = |Q|$, which indicates $OPT_{[\tau'+1,\tau'+U]} > \lambda$ as before. ∎

Theorem 2: Algorithm 1 is $\mathcal{O}(\ln n)$ competitive.

Proof: Lemma 2 shows that $\frac{1}{\lambda} < OPT_{[\tau'+1,\tau'+U]}$ in terms of the final λ. If $OPT = OPT_{[\tau'+1,\tau'+U]}$, we directly

Algorithm 2: Pruning Low Edge Loads

input : The statistics $\widehat{l}(e,\tau)$ and $r(e,\tau)$ from history
output: A candidate set $\mathcal{C}$

1 Sort $\{(e,\tau)|e \in E, \tau = 0,1,\ldots,T\}$ in the descending order of $\widehat{l}(e,\tau) + r(e,\tau)$
2 $\mathcal{C} \leftarrow \emptyset$
3 **foreach** (e,τ) *in the sorted list* **do**
4 $\quad lb \leftarrow \max(lb, \widehat{l}(e,\tau) - r(e,\tau))$
5 $\quad$ **if** $\widehat{l}(e,\tau) + r(e,\tau) < lb$ **then**
6 $\quad\quad$ return $\mathcal{C}$
7 $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{(e,\tau)\}$

have $\frac{\lambda}{2} < OPT$. If $OPT = OPT_{[0,\tau']} > OPT_{[\tau'+1,\tau'+U]}$, we have $\frac{\lambda}{2} < OPT_{[\tau'+1,\tau'+U]} < OPT$. Combining Lemma 1, the maximum load is at most $\mathcal{O}(\lambda \ln n) = \mathcal{O}(\ln n) \cdot OPT$. ■

B. Online Routing with History

Based on SOR, we propose the algorithm SRH that further optimizes the performance by considering the future traffic status. Intuitively, if we know that some road segments are about to be crowded, we can avoid using the paths that will traverse them. Since the real traffic is often periodic (*e.g.*, in a cycle of 24 hours), we could obtain such information from history. Specifically, if we know that some edge-step pair (e,τ) cannot be the one with the maximum edge load with high probability (from the statistics), we do not need to use its corresponding variable $x(e,\tau)$ in the metric of selecting paths. By considering only those pairs that are most likely to be congested in the metric, we avoid using them and can further optimize the performance. We first generate a candidate set of such edge-step pairs (denoted by $\mathcal{C}$) by estimating the edge load and the cardinality of the query set, and then give the algorithm SRH with a better competitive ratio of $\mathcal{O}(\ln(|\mathcal{C}|))$.

1) Estimating the Edge Load: In the following, we consider each edge load $l(e,\tau)$ as a random variable and use $\widehat{l}(e,\tau)$ to denote its sample means in history.

The idea is that if the edge load $l(e,\tau)$ is smaller than any edge load $l(e',\tau')$ with high probability, we could prune the pair (e,τ) since we only care about the maximum load. We would keep the edge-step pairs that cannot be pruned in the set $\mathcal{C}$. To find such events of high probability, we first obtain a confidence interval for each load $l(e,\tau)$, defined by *lower* and *upper confidence bounds* (i.e., it lies in the interval with high probability). If the upper bound of the load of a pair (e,τ) is smaller than the lower bound of the load of some other pair (e',τ'), we know that $l(e,\tau) < l(e',\tau')$ with high probability.

Example 8: In Figure 5, we plot the confidence intervals for five pairs, with lower and upper bounds shown in bold points. The lower bound of the load $l(e_3,1)$ is shown by the dashed line with its y-axis value of $\widehat{l}(e_3,1) - r$, where r is the *confidence radius*, defined below. Since it is greater than the upper bound of $l(e_1,0)$, we can safely prune $(e_1,0)$.

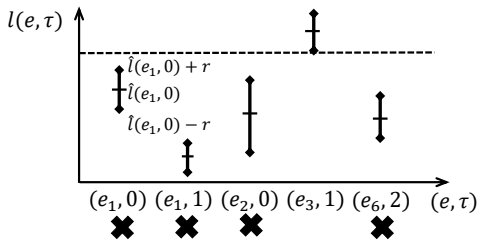


Fig. 5: The confidence bounds of edge loads

To define the confidence radius r, we first give the lower and upper bounds of $l(e,\tau)$. By the Hoeffding bound [33], we have $\mathbb{P}(|l(e,\tau) - \widehat{l}(e,\tau)| \geq \epsilon) \leq 2\exp\left(-\frac{2\epsilon^2 N}{R^2}\right)$, where ϵ, N, R are a non-negative real number in $(0,1)$, the number of samples, and the range of $l(e,t) \geq 0$, respectively. Note that all the samples are from the historical loads corresponding to the pair (e,τ) in cycles. Using an equivalent form, we have the following with probability $1-\delta/2$, $l(e,\tau) \in [\widehat{l}(e,\tau) - r(e,\tau), \widehat{l}(e,\tau) + r(e,\tau)]$, where the confidence radius $r(e,\tau) = \sqrt{\frac{R^2(\ln 4 - \ln \delta)}{2N}}$. This can be proved by replacing ϵ with $r(e,\tau)$ in the Hoeffding bound.

The pruning procedure is summarized in Algorithm 2. In Line 1, we first sort all the edge-step pairs in the descending order of their upper bounds $\widehat{l}(e,\tau) + r(e,\tau)$, and T is the number of steps in a cycle. In Line 4, we maintain a maximal lower bound lb when iterating (e,τ). If there exists a pair whose upper bound is lower than lb, we can stop the algorithm and safely prune all the remaining edge-step pairs (Lines 5–6) by Theorem 3. We initialize the candidate set $\mathcal{C}$ as an empty set in Line 2 and update it in Line 7.

Example 9: In Figure 5, suppose that we only need to consider the five pairs for simplicity. We first sort them by their upper bounds: $(e_3,1),(e_1,0),(e_2,0),(e_6,2),(e_1,1)$. After we process $(e_3,1)$, the lb is updated as $(e_3,1)$'s lower bound. When we process $(e_2,0)$, Algorithm 2 stops since its upper bound is lower than lb. Finally, $\mathcal{C} = \{(e_3,1)\}$.

Theorem 3: With probability $1-\delta$, any pair (e,τ) with $\widehat{l}(e,\tau) + r(e,\tau) < lb$ is not the one with the maximum load.

Proof: There must be one pair (e',τ') before the algorithm stops such that $lb = \widehat{l}(e',\tau') - r(e',\tau')$. By the Hoeffding bounds, we know that $l(e',\tau') > \widehat{l}(e',\tau') - r(e',\tau') = lb$ and $l(e,\tau) < \widehat{l}(e,\tau) + r(e,\tau) < lb$ both with probability $1-\delta/2$. After chaining these inequalities and using the union bound, we have $l(e,\tau) < l(e',\tau')$ with probability $1-\delta$. ■

2) Estimating the Cardinality of the Query Set: Instead of estimating the edge loads directly, we introduce the other useful statistics in pruning the edge-step pairs. Let Q^ι for $\iota \in \mathbb{N}$ denote the set of queries with their departure times in $[\iota, \iota+1)$. We consider the estimate $|\widehat{Q^\iota}|$.

The basic intuition is as follows. At each step ι, we maintain the current maximum load up to step ι, denoted by $L(\iota)$. Before processing the queries in Q^ι, we can ignore some edge-step pairs such that their loads are so small that they cannot be the maximum one at step $\iota+1$. Formally, $l(e,\iota+1)$ cannot

Algorithm 3: Spatiotemporal Routing with History (SRH)

1 $x(e,\tau) \leftarrow \frac{1}{2|\mathcal{C}|c(e))}$ and $v(e,\tau) \leftarrow 0$ for each (e,τ)
  $\lambda \leftarrow \min_e \frac{1}{c(e)}$
2 **for** *time step* $\iota \leftarrow 0, 1, \ldots$ **do**
3 $L(\iota) \leftarrow \max_{e,\tau} l(e,\tau)$
4 $\mathcal{C}' \leftarrow \mathcal{C}$ from Algorithm 2
5 **foreach** $e \in E$ **do**
6 **if** $l(e,\iota+1) + |\widehat{Q^\iota}| + r(Q^\iota) \leq L(\iota)$ **then**
7 $\mathcal{C}' \leftarrow \mathcal{C}' \setminus \{(e,\iota+1)\}$
8 **foreach** *new query* $q \in Q^\iota$ **do**
9 $\mathcal{P}(q) \leftarrow \{p | s_p = s_q, d_p = d_q, \gamma_p \leq (1+a)\gamma_{f^*(q)}\}$
10 $p \leftarrow \arg\min_{p \in \mathcal{P}(q)} \sum_{(e,\tau)\in\mathcal{C}'} \mathbb{1}_p(e,\tau) x(e,\tau)$
11 **while** $\sum_{(e,\tau)\in\mathcal{C}'} \mathbb{1}_p(e,\tau) x(e,\tau) > \lambda$ *or there exists one* $x(e,\tau) > \frac{\exp(\frac{1}{2})}{c(e)}$ **do**
12 $\lambda \leftarrow 2\lambda$
13 $x(e,\tau) \leftarrow \frac{(1+\frac{1}{2\lambda c(e)})^{v(e,\tau)}}{2|\mathcal{C}|c(e)}$ for all e,τ
14 $p \leftarrow \arg\min_{p \in \mathcal{P}(q)} \sum_{(e,\tau)\in\mathcal{C}'} \mathbb{1}_p(e,\tau) x(e,\tau)$
15 use the path p for query q, i.e., $f(q) \leftarrow p$
16 **foreach** e,τ *such that* $\mathbb{1}_{f(q)}(e,\tau) = 1$ **do**
17 $v(e,\tau) \leftarrow v(e,\tau) + 1$
18 $x(e,\tau) \leftarrow (1 + \frac{1}{2\lambda c(e)}) x(e,\tau)$

be the maximum load at step $\iota+1$ if the load is smaller than the current maximum load $L_{[0,\iota)}$ even when all the paths that return for Q^ι traverse $(e,\iota+1)$, i.e., $l(e,\iota+1)+|Q^\iota| \leq L_{[0,\iota)}$. Finally, we just need to use the upper confidence bound of $|Q^\iota|$ to replace it as in the previous technique. Note that this procedure has to be done at each step ι.

3) Optimized Online Routing: The two techniques above are summarized in Algorithm 3. The basic procedure is similar to Algorithm 1. We can imagine that there are Um candidate pairs in Algorithm 1 (i.e., $\mathcal{C} = \{(e,\tau) | e \in E, \tau = \iota+1, \ldots, \iota+U\}$) and it uses the pairs whose location indicators are equal to 1. Similarly, Algorithm 3 uses the edge-step pairs in $\mathcal{C}$ whose location indicators are equal to 1. The only three differences are that we first generate $\mathcal{C}$ by Algorithm 2, that we apply the technique in Section III-B2 in Lines 3–7, and that we replace Um by $|\mathcal{C}|$ in Lines 1 and 13 and change the sum of variables in Lines 10 and 14. The time complexity is still $\mathcal{O}(m\ln(n))$.

Example 10: Suppose that $\mathcal{C} = \{(e_3,1)\}$. All the variables are initialized as $\frac{1}{2|\mathcal{C}|c(e)} = \frac{1}{2}$. Algorithm 3 would select the path (e_1) for the first two queries $q_1 = q_2 = (0, A, B)$. The incurred loads are also shown by the white blocks in Figure 3b. This is because the path (e_1) contains no pair in $\mathcal{C}$ and its sum of variables is 0, but the metric value for the path (e_2,e_3,e_4) is $\frac{1}{2}$ since $\mathbb{1}_{(e_2,e_3,e_4)}(e_3,1) = 1$. For the two queries $q_3 = q_4 = (0.1, C, D)$, we can only select (e_5,e_3,e_6). The variable $x(e_3,1)$ is first updated as $(1+\frac{1}{\lambda c(e)})\frac{1}{2} = 3/4$ and then $(1+\frac{1}{\lambda c(e)})\frac{3}{4} = 9/8$ after the query q_4. Algorithm 3 results in an optimal maximum load $l(e_3,1) = 2$ in this example.

Theorem 4: Algorithm 3 achieves a competitive ratio of $\mathcal{O}(\ln|\mathcal{C}|)$ with probability $1-\delta$.

Proof: We can construct similar primal-dual programs by using $\mathcal{C}$ instead of all (e,τ) for $e \in E$ and $\tau'+1 \leq \tau \leq \tau'+U$. Since we prune the candidate pairs correctly with probability $1-\delta$ as discussed above, we regard those events as deterministic ones in the following. We construct the first program with constraints only for those $(e,\tau) \in \mathcal{C}$ and the second one with the sum of terms w.r.t. $(e,\tau) \in \mathcal{C}$ in the constraints. The initial primal objective value also starts with $1/2$. We can get similar results to Lemma 1 and Lemma 2. Since the maximum load occur in $\mathcal{C}$ with high probability, we still have $\lambda/2 < OPT$ and $l(e,\tau) = \mathcal{O}(\ln(|\mathcal{C}|)) \cdot OPT$ ∎

IV. Dynamic Edge Weights

When each driver follows the route that the algorithms returned for her past routing query, the edge weight can change at any time in a dynamic network [9], [24] and also affect her route. We can handle it by canceling the effects of the past query and processing a new query with her current location and original destination as the new query's source and destination, respectively. The new query just acts like the past one does not exist. The main idea is to build an index to efficiently find the affected past queries and maintain the variables and the loads correctly.

Two types of queries are not affected when the weight $w_{e'}$ of an edge e' is updated at some time $t' \in \mathbb{R}_{\geq 0}$. The first is the future query since our SOR and SRH all find routes based on the current updated weights. The second is the past query that will not traverse the updated edge e' after the time t'. The rest are the affected queries which are currently traversing e' or will traverse it.

To efficiently find those affected queries, we build an index which stores the mapping from edges to the paths routed so far, denoted by $trav(e)$. Specifically, for each edge e, we store the triple (p, t_{dep}, t_{arr}) where $t_{dep} \in \mathbb{R}_{\geq 0}$ is the departure time of the path p and $t_{arr} \in \mathbb{R}_{\geq 0}$ is the time when the path p finishes traversing the edge e. We insert these triples into $trav(e)$ after the routing algorithms return the path p for each query. We can simply iterate the edges in the path p and compute the time t_{arr} of each edge. Note that if the weight is so small that the path p traversing the edge does not show at any step, we can omit the related triples (as e_5 in Example 4) since no loads and variables will be affected.

Example 11: Back to Example 5, suppose that we use SRH to return paths. At $t=0$, after we return $p_1 = (e_1)$ and $p_2 = (e_1)$ to query q_1 and q_2, respectively, we add $(p_1, 0, 1.3)$ and $(p_2, 0, 1.3)$ to $trav(e_1)$ since p_1 and p_2 finish traversing e_1 at $t = 1.3$. The procedure is also shown in Figure 6. At $t = 0.1$, for query q_3 and q_4, we similarly add $(p_3, 0.1, 1.3)$ and $(p_4, 0.1, 1.3)$ into $trav(e_3)$ and $(p_3, 0.1, 2.3)$ and $(p_4, 0.1, 2.3)$ into $trav(e_6)$. We add no triple to $trav(e_5)$ because w_{e_5} is so small that p_3 and p_4 traversing e_5 do not show at any step.

Algorithm 4: Edge Weight Update

input : The old and new weights for the updated edge e', the update time t', and the index $trav(e)$
output: The index $trav(e)$

1 **foreach** $(p, t_{dep}, t_{arr}) \in trav(e')$ **do**
2 **if** $t_{arr} < t'$ **then**
3 remove the triple and continue
4 cancel the effects of the path p on the variables $x(e,\tau)$, the loads $l(e,\tau)$, and the index $trav(e)$
5 route a new path for this affected query with its new source based on the new edge weight
6 update the index, variables, and loads

Now suppose that the weight w'_e of an edge e' is updated at time $t' \in \mathbb{R}_{\geq 0}$. To obtain the affected queries in $trav(e')$, we remove the triples (p, t_{dep}, t_{arr}) in $trav(e')$ with $t_{arr} < t'$ since the path has traversed e' at time t'. For these affected (p, t_{dep}), what we need to do is to cancel their effects on the variables $x(e,\tau)$, loads $l(e,\tau)$, and the index of $trav(e)$, then reroute the paths by regarding them as new queries with the new sources and departure times and the original destinations, and finally maintain the variables, the loads, and the index by the newly returned path. To cancel the effect on $trav(e)$, we iterate p's edges based on the original weight and remove the corresponding triples in the index. To cancel the effect on the loads and variables, we iterate the edge-step pairs that p traverses. Finally, we maintain the index, variables, and loads index as previously stated.

Note that p's current location could lie on any edge. For ease of computation, we can use the destination vertex of this edge as the new source of the new query. We estimate its departure time at the new source by using the current time plus the remaining time of traversing this edge. If its current location lies on the updated edge, we can multiply the remaining time by the ratio of the new edge weight to the old one.

Algorithm 4 summarizes the procedure. For each triple in $trav(e')$, we first prune the obsolete ones in Lines 2–3. We cancel the effects of the corresponding path in Line 4, return a new path based on the new edge weight in Line 5, and finally update the index, variables, and loads in Line 6.

Example 12: At $t = 1.1$, suppose that w_{e_3} is increased to 2. We first prune the triples with $t_{arr} < 1.1$ in $trav(e_3)$. For $(p_3, 0.1, 1.3)$, we cancel its effect on the index $trav(e)$ by removing $(p_3, 0.1, 1.3)$ from $trav(e_3)$ and $(p_4, 0.1, 2.3)$ from $trav(e_6)$. We then route $q_3 = (1.5, F, D)$ with its new source as the destination of e_3. Its new departure time is $1.1 + 0.2 \cdot 2/1 = 1.5$, where the remaining time on e_3 is $0.1 + 0.2 + 1 - 1.1 = 0.2$. Since we return the path $p_5 = (e_6)$ to this new query, we update $trav(e_3)$ by inserting $(p_5, 1.5, 2.5)$ with its $t_{arr} = 1.5 + 1 = 2.5$. The updates for loads and variables are omitted. We similarly process $(p_4, 0.1, 1.3)$.

The space cost is $\mathcal{O}(|Q|U)$, where U is the maximum time span. This is because each query can be stored in different $trav(e)$ at most U times. However, it is much smaller in practice since we remove obsolete triples continuously. The time complexity is $\mathcal{O}(U \max_e trav(e))$ since we examine each triple in $trav(e)$ and for each triple, the time of canceling and updating the index, the variables, and loads are all $\mathcal{O}(U)$.

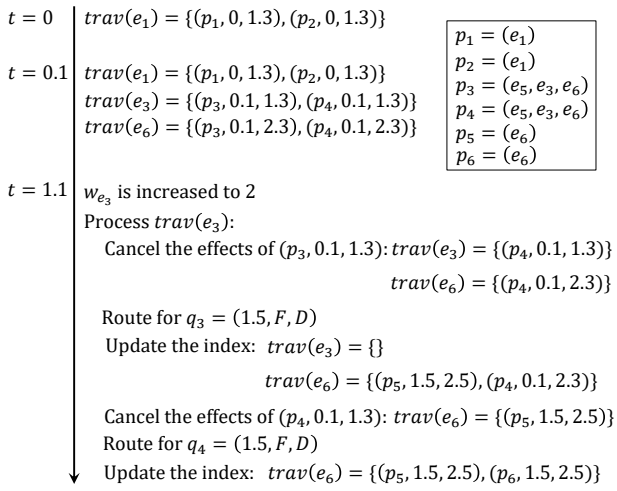


Fig. 6: Edge weight update

TABLE I: Real dataset statistics

Datasets	$\|Q\|$	Time Periods
NYC	20,775	5 p.m. - 6 p.m.
Baidu Maps	22,881	7 a.m. - 7:30 a.m.
Didi	22,794	9 a.m. - 9:35 a.m.

V. Experimental Study

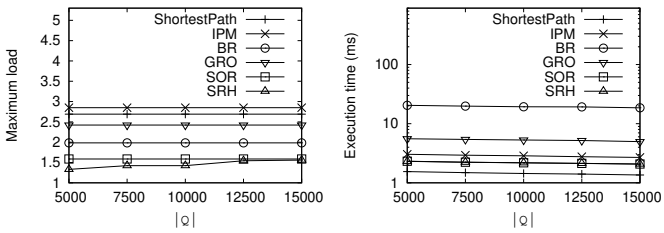
A. Experiment Setup

In our experiments, algorithms are implemented using the compiler gcc 9.4.0 with O3 optimization and performed on a machine with a 2.66GHz CPU and 48GB RAM installed with the CentOS 7 Linux distribution.

Datasets. We collected road network and query data for experiments. We used two road networks, one in New York City (NYC) from DIMACS[1] and the other in Beijing (BJ) from OpenStreetMap[2]. The NYC network has 264,346 nodes and 733,846 edges, and the Beijing network has 188,229 nodes and 436,648 edges. For routing queries, we used NYC taxi trip data from August 2013 from the NYC TLC Trip Record Data [34], routing query data from April 2017 from Baidu Maps's Q-Traffic Dataset [35], and online taxi-calling trip data from October 2016 from Didi [36]. The NYC queries were tested on the NYC network, and the queries from Baidu Maps and Didi were tested on the Beijing network. We set the duration of a time step to one minute according to the setup in Baidu Maps [25] so that $t = 0.1$ means 6 seconds from now.

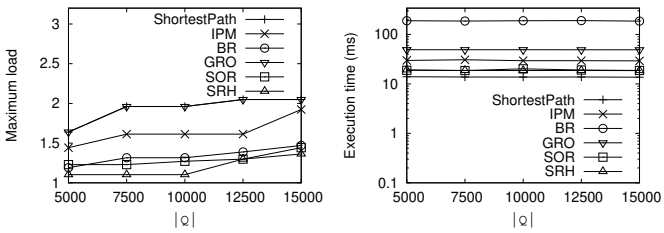
We evaluated our algorithms by varying (1) the query set Q, (2) the detour factor a, and (3) the penetration rate ρ (defined as the ratio of the number of queries that adopt our proposed/existing routing algorithms to the total number of

[1]http://www.dis.uniroma1.it/challenge9/download.shtml
[2]https://download.bbbike.org/osm/bbbike/Beijing/



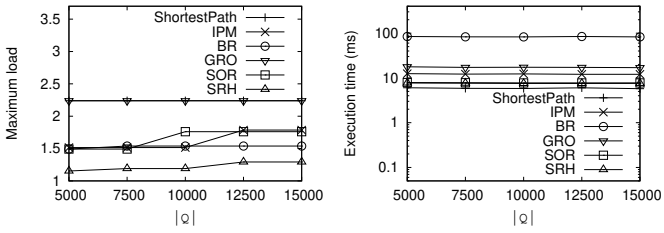
(a) Max. load when varying $|Q|$ (b) Time when varying $|Q|$

Fig. 7: Varying $|Q|$ on NYC



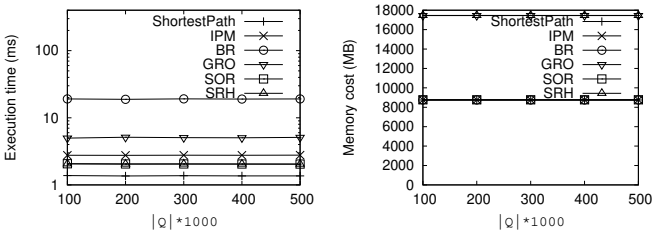
(a) Max. load when varying $|Q|$ (b) Time when varying $|Q|$

Fig. 8: Varying $|Q|$ on Baidu



(a) Max. load when varying $|Q|$ (b) Time when varying $|Q|$

Fig. 9: Varying $|Q|$ on Didi



(a) Time when varying $|Q|$ (b) Memory when varying $|Q|$

Fig. 10: Scalability test

queries). (1) Following existing work [9], [24], [37], we set $|Q| = [5000, 7500, \underline{10000}, 12500, 15000]$, where the default value is underlined. We extracted the query sets of the above sizes from three sources shown in Table I. We also conducted the scalability test by varying $|Q| = [1, 2, 3, 4, 5] \times 10^5$ on NYC. (2) For the detour factor a, since a small value could result in less detour cost, we varied a small $a = [\underline{0.05}, 0.075, 0.1, 0.125, 0.15]$ from the three sources but also tested large values from 0.1 to 0.5. Note that SRH uses a set $\mathcal{C}$ of candidates from history to guide its routing. For each of the three sources, we generate and use only one set $\mathcal{C}$ of candidate pairs by considering the corresponding edges and periods in the previous 20 days. (3) For the penetration rate, we vary $\rho = [0.2, 0.4, 0.6, 0.8, \underline{1}]$ to simulate the real scenarios where the ρ fraction of vehicles follow the recommended routes. In our experiment, the $1-\rho$ fraction of queries adopt ShortestPath (instead of our proposed/existing algorithm) to find routes. All the experiments use the same setting of $\delta = 0.1$, which is the best value after multiple tests.

Compared algorithms. We compared four algorithms widely used for practical routing.

(1) ShortestPath. It regards the travel time of each road segment as the edge weight and finds the path minimizing the sum of traversed edges' weights.

(2) Iterative Penalty Method (IPM). The representative approach of alternative paths (though not for minimizing congestion) is the IPM which gives alternative paths by imposing a penalty weight on the traversed edges [10]. Since the recent study shows that the variant using a penalty weight of 1.4 is superior to competitors [16], we implement this variant.

(3) Balanced Routing (BR). BR is one congestion avoidance strategy in the transportation area [18]. It defines the entropy-based metric function to choose the best path from a set of dissimilar paths [17].

(4) Global Routing Optimization (GRO). GRO is the state-of-the-art routing strategy for queries in batches [23]. It finds routes for a given batch of queries by gradually reducing the sum of travel times. Since our online queries should be processed right away but not in a batch, we implement its variant by combining the queries of each step in a batch.

(5) SOR. Our first algorithm uses no history and chooses the path based on the existing congestion status.

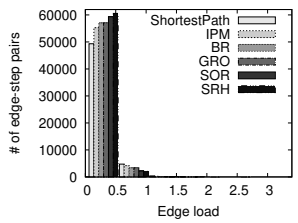
(6) SRH. Our second improvement generates future congested edge-step pairs from history to guide the routing.

We assess the performance of the above algorithms in terms of the maximum load and the average execution time per query. Basically, different load values can be shown in different colors (*e.g.*, red and green) of road segments in navigation apps. A load less than one for an edge means that there is no congestion and that all vehicles can traverse the edge at a given normal speed at the time corresponding to the step [26]–[28].

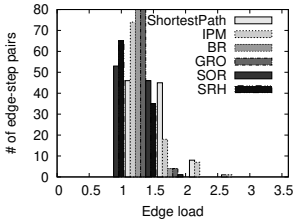
B. Experiment Results

1) Effect of $|Q|$: Figures 7, 8, and 9 show the results of varying $|Q|$ on NYC, Baidu, and Didi, respectively.

For the maximum load, all the algorithms have larger values when $|Q|$ is larger. Note that some lines are flat because the datasets of large $|Q|$ are made up by supplementing queries to the datasets of small $|Q|$ and the maximum load appears early in the datasets of small $|Q|$ (*e.g.*, the maximum load of ShortestPath appear in the first 5,000 queries on NYC). Among all the algorithms, *ShortestPath* is the worst since it does not consider the congestion status of the edges. *IPM* achieves larger maximum loads on NYC and Baidu but smaller ones on Didi. *BR* is basically at the medium level. *GRO* performs worse on Baidu and Didi since the batch in one minute gives it little information, and it cannot handle fast online queries. *SOR* is

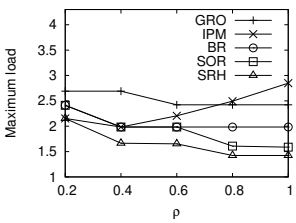


(a) All edge-step pairs

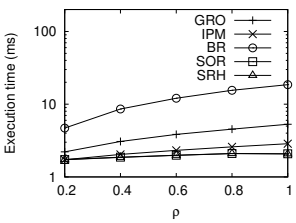


(b) Top 100 edge-step pairs

Fig. 11: Results of load distribution on NYC



(a) Max. load when varying ρ



(b) Time when varying ρ

Fig. 12: Results of varying the penetration rate ρ (NYC)

the second best one in some cases. *SRH* always achieves the smallest maximum load and could reduce the maximum load of the baseline by nearly 20% on Didi.

In terms of the execution time, all the algorithms have a constant time cost per query. Their times are insensitive to $|Q|$ because each query is processed by each call of the algorithms independently and because we report the average time per query. *BR* is slow on all datasets because it needs to compute and compare multiple paths for each query. The time costs of the other algorithms are around 10 ms and hence are acceptable in practice.

2) Load distribution: Figure 11 shows the load distribution on the default setting. We split edge-step pairs into equal-width intervals by their edge loads and count the number of pairs for each interval. Note that we omit the pairs with zero loads.

Figure 11a shows the load distribution of all pairs. We see that the loads for most pairs are less than 0.5. We need to focus on the tiny minority of pairs with loads higher than 1 because the most congested ones are bottlenecks for congestion minimization. We consider the top 100 edge-step pairs with the highest loads in Figure 11b. We find that the loads for the top 100 pairs basically exceed 1. The pairs of *SOR* and *SRH*, shown in dark black bars, concentrate on the left interval [1,1.5]. However, *ShortestPath* and *IPM* could have loads above 2. The results are consistent with the least maximum load of *SRH*.

3) Scalability test.: Figure 10 gives the results of the scalability test where we vary $|Q|$ in the order of 10^5. The average processing time of all the algorithms except *BR* is still low in around 10 ms per query. For the memory cost, they are stable since the dominant part of memory consumption lies in the edge loads, where we have to store values for each edge-step pair. Note that the memory cost is in the same order of the problem size (which is $\mathcal{O}(mT)$) since we need to check the edge load of each edge-step pair. *SRH* takes double the memory of the other ones because when checking if a pair is in the candidate set $\mathcal{C}$, we simply use Boolean values for all pairs, with the same size of $\mathcal{O}(mT)$ as the edge loads. In conclusion, all the algorithms scale with respect to $|Q|$.

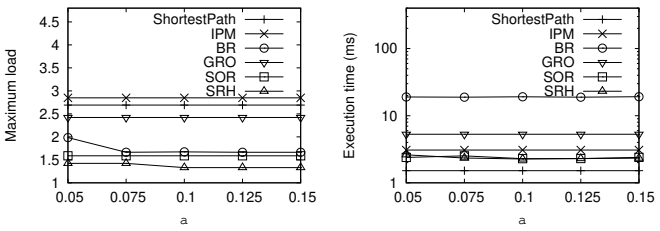
4) Effect of ρ: The results of varying the penetration rate ρ is shown in Figure 12. In Figure 12a, we see that all algorithms, except IPM that is not designed for congestion minimization, tend to achieve lower maximum loads when the penetration rate is higher, which also indicates that higher usage of the routing algorithms is beneficial for mitigating congestion. We also observe that SRH achieves the smallest maximum load among all algorithms. In terms of the query time, all algorithms need more time when ρ is larger because more queries adopt them, yielding higher time costs.

5) Effect of small a: Figures 13, 14, and 15 present the results of varying a small detour factor a on NYC, Baidu, and Didi, respectively. In terms of the maximum load, *ShortestPath*, *IPM*, and *GRO* remain unchanged since they do not consider the detour constraint. We see a downward trend for the other three algorithms when a is increased. This is because we allow more detour costs and give the algorithms more path choices. For the maximum load when $a = 0$, *SOR* and *SRH* achieve the same load as *ShortestPath* since they are not allowed to use detours. Setting a small a reduces the load significantly (e.g., the load of SRH changes from 2.69 to 1.42 when a is increased to 0.05 on NYC), which is also our motivation for sacrificing small detour costs to mitigate congestion. It is possible that there is little improvement when a is increased. The main reason is that the load has decreased significantly when a becomes non-zero, and there is a point where decreasing a makes no difference due to the existence of "bottleneck" edges. They always appear in the paths for some queries, no matter how we set a (*e.g.*, several queries may have the same source with only one incident edge.). *SRH* still achieves the smallest maximum load among all algorithms on all datasets. The execution times of our algorithms are stable. The main reason is that for most queries, the paths minimizing the metric function do not violate the constraint and can be found quickly.

6) Effect of large a: To further explore the effect of a large detour factor, we show the results of a from 0.1 to 0.5 in Figure 16. For the maximum load, it could be found that only *SRH* gradually decreases when a is larger. There is a plateau for *SOR* and *BR* after some point. The reason is similar to the previous one of the bottleneck edges. We can learn that a small detour factor of around 0.1 is enough for congestion minimization. Large values may not give noticeable improvement. The rest findings are similar.

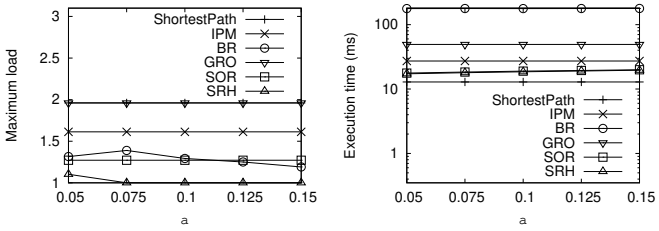
C. Dynamic edge weights

We evaluate Algorithm 4 by using the default settings on NYC. Following existing work [7], [24], we randomly select 1000 edges and change their weights w_e by drawing samples from a uniform distribution on $[w_e, (1+c)w_e]$, where $c \in \{1, 2, 3, 4, 5\}$. The update times of weights are uniformly



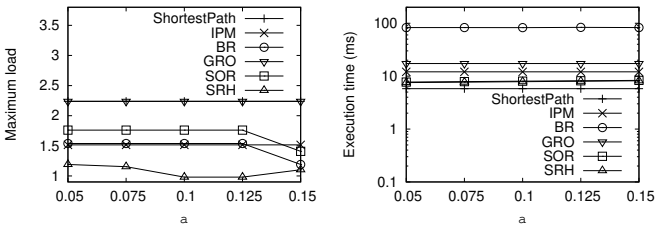
(a) Max. load when varying a (b) Time when varying a

Fig. 13: Varying a on NYC



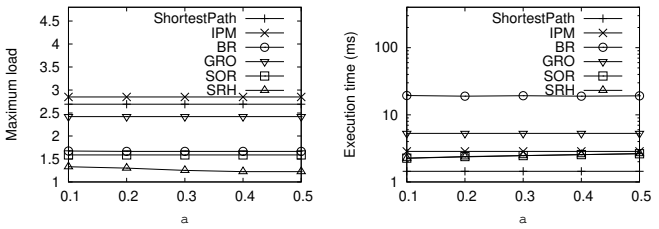
(a) Max. load when varying a (b) Time when varying a

Fig. 14: Varying a on Baidu



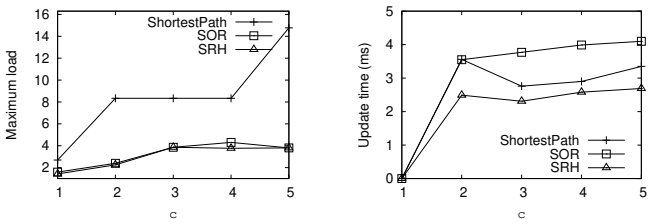
(a) Max. load when varying a (b) Time when varying a

Fig. 15: Varying a on Didi



(a) Max. load when varying a (b) Time when varying a

Fig. 16: Varying a large a



(a) Max. load when varying c (b) Time when varying c

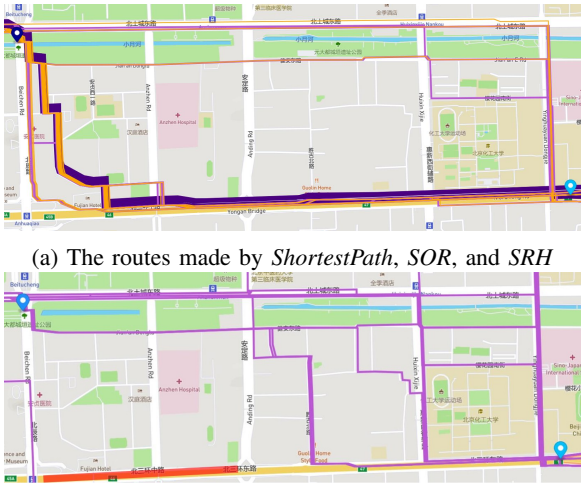
Fig. 17: Results of edge weight updates (NYC)

distributed over 20 minutes. Since *IPM*, *BR*, and *GRO* cannot be adapted easily to handling dynamic weights, we only implement variants of *ShortestPath*, *SOR*, and *SRH*.

The results are shown in Figure 17. For the maximum load, when c increases, the upper bound of the edge weights increases. Although the updated edges may not be the most congested ones, the maximum loads of all algorithms generally increase. The time cost per update is 0 when c is 0 since no weight is updated. All the algorithms perform one edge weight update within 5 ms, which is competitive to the update time of existing solutions [7], [24].

D. Case Study

We study the traffic status of an area next to the North 3rd Ring Road in Beijing. To simulate the traffic, we duplicate the queries from 7:00 a.m. to 7:30 a.m. (during the morning rush hour) on April 21, 2017 from Baidu Maps five times. The routes of the *ShortestPath*, *SOR*, and *SRH* are shown in Figure 18a as dark purple, orange, and pink lines, respectively. It can be seen that *ShortestPath* chooses the path with the shortest travel time for 20 vehicles, which results in the most crowded road segments. SOR distributes some vehicles to other road segments but still guides many vehicles to the segments in the left part. *SRH* scatters the vehicles and guides 9 vehicles to use other road segments to reduce congestion. The



(a) The routes made by *ShortestPath*, *SOR*, and *SRH*

(b) The routes of *SRH* under a traffic change

Fig. 18: A case study of the Beijing network

recommended routes with limited detour costs are reasonable under our small detour factor setting.

For the same setting, we also simulate the scenario where the travel time of the segment in red increases, as shown in Figure 18b. We observe that *SRH* (shown in pink lines) can handle such a change by avoiding using the red segment.

E. Summary

(i) Our proposed *SRH* that considers future congestion status outperforms the state-of-the-art baselines. It can reduce up to 33% the maximum load incurred by baselines.

(ii) *SOR* and *SRH* are efficient in terms of the average processing time (around 10 ms per query). They are scale to large city road networks and frequent online routing queries and process each edge weight update with running times similar to those of existing solutions.

(iii) The proposed *SRH* performs well in real-world scenarios and responds to traffic changes immediately.

VI. Related Work

A. Routing for Shortest Travel Time

The basic algorithm for finding the shortest paths on road networks is Dijkstra's algorithm [38]. State-of-the-art solutions construct precomputed indexes to improve query efficiency. The indexes can be used to either prune the search space [39]–[41] or store distances as labels for quick lookups [7], [8], [42]–[44]. To handle real-time updates of edge weights, some studies directly reduce the update time complexity [9], [24], [37], [45], while other studies model edge weights by predictable time-dependent functions [46]–[53]. Since SOR and SRH essentially find the paths that minimize a metric function under a detour constraint, it is also a constrained shortest path, which has been studied widely [31], [54]–[59]. However, these existing proposals can cause congestion when they are used for answering a number of queries.

A line of work analyzes the inefficiency caused by routing algorithms that optimize the travel time or distance only. This is first formulated by the Price of Anarchy (PoA), defined as the ratio of the congestion cost over that of a globally optimal strategy [60]. Its worst case is analyzed [3], and subsequent studies consider the model under different cases: differentiable and convex latency functions [61], the heavy traffic [62], or both light and heavy traffic [5]. However, these studies assume that pairs of origins and destinations are given beforehand and try to analyze the inefficiency under different traffic models; they do not aim to influence and guide drivers. Our problem is to process online routing queries so as to guide drivers.

B. Routing for Alternative Paths

Finding alternative routes, in contrast to the shortest ones, is a possible way of reducing congestion. Early methods include Yen's algorithm [63] and Eppstein's algorithm [64]. Recent techniques are based on penalties [10], [11], plateaus [12], [13], and dissimilarity [14], [15], [65] (see [16] for a thorough study). However, one main issue is that there is no common notion of a "good" route. Their recommended routes may not optimize the congestion status and hence are suboptimal for the problem we study. Our proposed algorithms mitigate congestion by considering the existing and future traffic statuses.

C. Routing for Congestion Minimization

In the transportation area, congestion avoidance strategies on a small part of a network are considered, such as at junctions (see [18] for a summary). Some studies propose rerouting heuristics that change routes when certain congested conditions are met [17], [19], and others consider drivers as individual agents and study their interactions [20], [21]. These studies of ten involve detailed models of traffic elements, which reduces their scalability (as shown in our experiments). Experiments are often reported on networks with fewer than 10,000 edges. In contrast, our solution can process queries efficiently on large networks and can recommend routes based on existing and future traffic statuses. Moreover, recent studies consider minimizing the sum of travel times of queries given batches of queries [22], [23]. They optimize routes repeatedly and gradually when they are aware of many queries. However, they are effective only when they have sufficient query information. In real scenarios, online queries must be processed independently and immediately to meet the need for fast responses. They cannot wait for the algorithms to process the batches. Further, the returned paths may involve long detours, which are impractical since they do not support detour constraints. In contrast, our solutions are able to respond to frequent routing queries right away, and all returned paths satisfy a specified detour constraint.

In telecommunication networks, online algorithms are also proposed to improve congestion [32], [66]–[69]. Specifically, the routes in their problems increase the loads of traversed edges at an instant and last forever, whereas a vehicle in our problem can move along routes in both space and time. In one study [66], an optimal $\mathcal{O}(\ln n)$-competitive algorithm is proposed to minimize the maximum load. It is later proved to be equivalent to a primal-dual algorithm [32]. Other objectives can be to minimize the average latency [68], maximize the total throughput [67], or minimize the load under a different input model [69]. However, their networks are different from our road networks, and we are planning routes for vehicles that occupy different roads at different times. We consider practical issues such as the detour constraint and return routes based on the existing and future traffic statuses.

VII. Conclusion

This paper studies a new type of routing queries aiming at mitigating traffic congestion. To do this, we identify the load of each road segment (incurred by the vehicles following the recommended routes) in the temporal dimension. We first propose the Congestion-mitigating Spatiotemporal Routing (CSR) problem and prove its NP-hardness under the offline setting. To address it, we design the algorithm SOR based on the existing traffic status with a competitive ratio of $\mathcal{O}(\ln n)$, where n is the number of nodes. Noticing the recurring property of traffic congestion, we further optimize SOR by focusing more on future congested segments and propose SRH with a ratio of $\mathcal{O}(\ln |\mathcal{C}|)$, where $\mathcal{C}$ is a set of candidate congested segments derived from history. We report on experiments on real-world data that offer evidence that the proposed algorithms are capable of outperforming baselines and advancing the state-of-the-art. In future work, it is of interest to study how to minimize the average traffic status or how to optimize different traffic elements (e.g., controlling traffic light times or deciding where to build new roads). It is also of interest to consider the influence of external factors that cause unexpected congestion.

Acknowledgements

We are grateful to the anonymous reviewers for their constructive comments on this paper. This work was supported in part by PRP/026/21FX and the Innovation Fund Denmark centre, DIREC.

REFERENCES

[1] "Longest traffic jam," 2017, https://www.guinnessworldrecords.com/world-records/461618-longest-traffic-jam-number-of-vehicles.

[2] S. Davis and S. Diegel, "Transportation energy data book: Edition 22," *DIANE Publishing*, 2019.

[3] T. Roughgarden and É. Tardos, "How bad is selfish routing?" *J. ACM*, 2002.

[4] J. R. Correa, A. S. Schulz, and N. E. S. Moses, "Selfish routing in capacitated networks," *Mathematics of Operations Research*, 2004.

[5] R. Colini-Baldeschi, R. Cominetti, P. Mertikopoulos, and M. Scarsini, "When is selfish routing bad? the price of anarchy in light and heavy traffic," *Operation Research*, 2020.

[6] L. Li, M. Zhang, W. Hua, and X. Zhou, "Fast query decomposition for batch shortest path processing in road networks," in *ICDE*, 2020.

[7] Z. Chen, A. W. Fu, M. Jiang, E. Lo, and P. Zhang, "P2H: efficient distance querying on road networks by projected vertex separators," in *SIGMOD*, 2021.

[8] D. Ouyang, L. Qin, L. Chang, X. Lin, Y. Zhang, and Q. Zhu, "When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks," in *SIGMOD*, 2018.

[9] V. J. Wei, R. C. Wong, and C. Long, "Architecture-intact oracle for fastest path and time queries on dynamic spatial networks," in *SIGMOD*, 2020.

[10] V. Akgün, E. Erkut, and R. Batta, "On finding dissimilar paths," *Eur. J. Oper. Res.*, 2000.

[11] D. Cheng, O. Gkountouna, A. Züfle, D. Pfoser, and C. Wenk, "Shortest-path diversification through network penalization: A washington DC area case study," in *SIGSPATIAL*, 2019.

[12] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Alternative routes in road networks," *ACM J. Exp. Algorithmics*, 2013.

[13] L. Li, M. A. Cheema, M. E. Ali, H. Lu, and D. Taniar, "Continuously monitoring alternative shortest paths on road networks," *PVLDB*, 2020.

[14] H. Liu, C. Jin, B. Yang, and A. Zhou, "Finding top-k shortest paths with diversity," *IEEE Trans. Knowl. Data Eng.*, 2018.

[15] T. Chondrogiannis, P. Bouros, J. Gamper, U. Leser, and D. B. Blumenthal, "Finding k-shortest paths with limited overlap," *VLDB J.*, 2020.

[16] L. Li, M. A. Cheema, H. Lu, M. E. Ali, and A. N. Toosi, "Comparing alternative route planning techniques: A comparative user study on melbourne, dhaka and copenhagen road networks," *IEEE Trans. Knowl. Data Eng.*, 2021.

[17] R. Liu, H. Liu, D. Kwak, Y. Xiang, C. Borcea, B. Nath, and L. Iftode, "Balanced traffic routing: Design, implementation, and evaluation," *Ad Hoc Networks*, 2016.

[18] S. E. Hamdani and N. Benamar, "A comprehensive study of intelligent transportation system architectures for road congestion avoidance," in *UNet*, 2017.

[19] J. Pan, I. S. Popa, K. Zeitouni, and C. Borcea, "Proactive vehicular traffic rerouting for lower travel time," *IEEE Transactions on vehicular technology*, 2013.

[20] S. Wang, S. Djahel, and J. McManis, "A multi-agent based vehicles re-routing system for unexpected traffic congestion avoidance," in *ITSC*, 2014.

[21] P. Desai, S. W. Loke, A. Desai, and J. Singh, "Caravan: Congestion avoidance and route allocation using virtual agent negotiation," *IEEE Transactions on Intelligent Transportation Systems*, 2013.

[22] K. Li, L. Chen, and S. Shang, "Towards alleviating traffic congestion: Optimal route planning for massive-scale trips," in *IJCAI*, 2020.

[23] Y. Xu, L. Li, M. Zhang, Z. Xu, and X. Zhou, "Global routing optimization in road networks," in *ICDE*, 2023.

[24] D. Ouyang, L. Yuan, L. Qin, L. Chang, Y. Zhang, and X. Lin, "Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees," *PVLDB*, 2020.

[25] "Taffic api of baidu maps," 2022, https://lbsyun.baidu.com/index.php?title=webapi/traffic.

[26] A. M. Rao and K. R. Rao, "Measuring urban traffic congestion-a review." *IJTTE*, vol. 2, no. 4, 2012.

[27] R. L. Bertini, "You are the traffic jam: an examination of congestion measures," in *The 85th annual meeting of transportation research board*, 2006, p. 115.

[28] R. Arnott and K. Small, "The economics of traffic congestion," *American scientist*, vol. 82, no. 5, pp. 446–455, 1994.

[29] "Level of service," 2022, https://en.wikipedia.org/wiki/Level_of_service_(transportation).

[30] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[31] W. M. Carlyle and R. K. Wood, "Near-shortest and k-shortest simple paths," *Networks*, vol. 46, no. 2, pp. 98–109, 2005.

[32] N. Buchbinder and J. Naor, "Improved bounds for online routing and packing via a primal-dual approach," in *FOCS*, 2006.

[33] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American Statistical Association*, 1963.

[34] "NYC taxi trips," 2022, https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[35] B. Liao, J. Zhang, C. Wu, D. McIlwraith, T. Chen, S. Yang, Y. Guo, and F. Wu, "Deep sequence learning with auxiliary information for traffic prediction," in *KDD*, 2018.

[36] "Didi data," 2022, https://outreach.didichuxing.com/research/opendata/.

[37] Z. Yu, X. Yu, N. Koudas, Y. Liu, Y. Li, Y. Chen, and D. Yang, "Distributed processing of k shortest path queries over dynamic road networks," in *SIGMOD*, 2020.

[38] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[39] P. Sanders and D. Schultes, "Engineering highway hierarchies," *ACM J. Exp. Algorithmics*, vol. 17, no. 1, 2012.

[40] J. Dibbelt, B. Strasser, and D. Wagner, "Customizable contraction hierarchies," in *SEA*, 2014.

[41] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies," *Transp. Sci.*, 2012.

[42] T. Akiba, Y. Iwata, K. Kawarabayashi, and Y. Kawata, "Fast shortest-path distance queries on road networks by pruned highway labeling," in *ALENEX*, C. C. McGeoch and U. Meyer, Eds., 2014.

[43] W. Li, M. Qiao, L. Qin, Y. Zhang, L. Chang, and X. Lin, "Scaling up distance labeling on graphs with core-periphery properties," in *SIGMOD*, 2020.

[44] ——, "Scaling distance labeling on small-world networks," in *SIGMOD*, 2019.

[45] T. Hayashi, T. Akiba, and K. Kawarabayashi, "Fully dynamic shortest-path distance query acceleration on massive networks," in *CIKM*, 2016.

[46] E. Kanoulas, Y. Du, T. Xia, and D. Zhang, "Finding fastest paths on A road network with speed patterns," in *ICDE*, 2006.

[47] L. Li, S. Wang, and X. Zhou, "Time-dependent hop labeling on road network," in *ICDE*, 2019.

[48] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou, "Efficient route planning on public transportation networks: A labelling approach," in *SIGMOD*, 2015.

[49] Y. Wang, G. Li, and N. Tang, "Querying shortest paths on time dependent road networks," *PVLDB*, 2019.

[50] Y. Yuan, X. Lian, G. Wang, Y. Ma, and Y. Wang, "Constrained shortest path query in a large time-dependent graph," *PVLDB*, 2019.

[51] L. Li, W. Hua, X. Du, and X. Zhou, "Minimal on-road time route scheduling on time-dependent graphs," *PVLDB*, 2017.

[52] L. Li, S. Wang, and X. Zhou, "Fastest path query answering using time-dependent hop-labeling in road network," *IEEE Trans. Knowl. Data Eng.*, 2022.

[53] 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," *VLDB J.*, 2018.

[54] H. Joksch, "The shortest route problem with constraints," *Journal of Mathematical Analysis and Applications*, vol. 14, no. 2, pp. 191–197, 1966.

[55] G. Y. Handler and I. Zang, "A dual algorithm for the constrained shortest path problem," *Networks*, vol. 10, no. 4, pp. 293–309, 1980.

[56] D. Delling and D. Wagner, "Pareto paths with SHARC," in *SEA*, 2009.

[57] S. Storandt, "Route planning for bicycles - exact constrained shortest paths made practical via contraction hierarchy," in *ICAPS*, 2012.

[58] Z. Liu, L. Li, M. Zhang, W. Hua, P. Chao, and X. Zhou, "Efficient constrained shortest path query answering with forest hop labeling," in *ICDE*, 2021.

[59] L. Wang and R. C. Wong, "QHL: A fast algorithm for exact constrained shortest path search on road networks," *Proc. ACM Manag. Data*, vol. 1, no. 2, pp. 155:1–155:25, 2023.

[60] E. Koutsoupias and C. H. Papadimitriou, "Worst-case equilibria," in *STACS*, 1999.

[61] T. Roughgarden and É. Tardos, "Bounding the inefficiency of equilibria in nonatomic congestion games," *Games and Economic Behavior*, 2004.

[62] R. Colini-Baldeschi, R. Cominetti, and M. Scarsini, "On the price of anarchy of highly congested nonatomic network games," in *SAGT*, 2016.

[63] J. Y. Yen, "Finding the k shortest loopless paths in a network," *Management Science*, 1971.
[64] D. Eppstein, "Finding the k shortest paths," *SIAM J. Comput.*, 1998.
[65] Z. Luo, L. Li, M. Zhang, W. Hua, Y. Xu, and X. Zhou, "Diversified top-k route planning in road network," *PVLDB*, 2022.
[66] J. Aspnes, Y. Azar, A. Fiat, S. A. Plotkin, and O. Waarts, "On-line routing of virtual circuits with applications to load balancing and machine scheduling," *J. ACM*, 1997.
[67] A. Goel, M. R. Henzinger, and S. A. Plotkin, "Online throughput-competitive algorithm for multicast routing and admission control," in *SODA*, 1998.
[68] P. Harsha, T. P. Hayes, H. Narayanan, H. Räcke, and J. Radhakrishnan, "Minimizing average latency in oblivious routing," in *SODA*, 2008.
[69] N. K. Thang, "A competitive algorithm for random-order stochastic virtual circuit routing," in *ISAAC*, 2019.