Online Spatiotemporal Routing for Congestion Minimization

Paper ID: 320

ABSTRACT

Traffic congestion is a severe problem in modern cities. Since more drivers use navigation apps, we search to provide alternative routes to reduce congestion. However, routing in the presence of real traffic could be challenging. First, routing queries are received and processed in an online streaming way. The currently recommended routes may contradict what we should do for future queries. Second, the congestion happens at specific road segments and periods. We need to identify them correctly. In this paper, we propose the Spatiotemporal Routing Problem (SRP) which describes the traffic congestion scenario. Its objective is to minimize the maximum number of vehicles at some road segments and periods. Our first algorithm, called the Spatiotemporal Oblivious Routing (SOR) algorithm, is guaranteed to have at most $O(\ln n)$ the optimal value, where n is the number of nodes in the road network. By additionally utilizing the history to gain some foresight of congested roads, we propose our second algorithm called the Spatiotemporal Routing with History (SRH) algorithm with a better ratio of $O(\ln |C|)$, where C is a candidate set of potential congested roads. Extensive evaluations on three real-world datasets demonstrate the superiority of the SRH. It could respond to a query within 20 ms and reduce the number of vehicles on the most congested road by nearly 20%.

KEYWORDS

Routing Query, Congestion Minimization

1 INTRODUCTION

Nearly everyone in modern cities worldwide has experienced traffic congestion to some degree. For instance, daily rush hours can cause significant delays for commuters going to work or returning back home. In some extreme cases, drivers could be stuck in an expressway for days and move just two miles a day [1]. Apart from the inconvenience and frustration on motorists, it brings about a high financial cost on governments or some irreversible environmental damage due to the exhaust gas of idling vehicles on roads [18]. Therefore, reducing congestion is necessary in several aspects.

The development of mobile Internet has spawned many new routing applications, including GPS navigation 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). More and more drivers enjoy the fastest routes that these applications provide for their routing queries from an origin to a destination [4]. However, studies have shown that these routes could aggravate congestion since they are concerned more about their own shortest travel time, which are *selfish* behaviors compared with the traffic condition according to a global optimal routing strategy [15, 17, 48]. On the other hand, since drivers are relying more on the guidance from routing applications, we naturally raise the question: how could the congestion be ameliorated if the recommended routes (with acceptable detours) are slightly longer than the fastest ones in these routing applications?

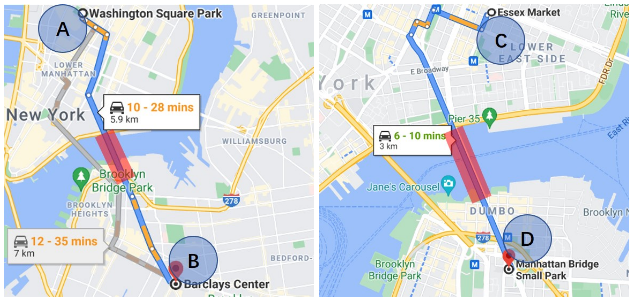
Routing to minimize congestion is non-trivial in real traffic. For one point, platforms receive routing queries in an online streaming way which requires us to respond to each query immediately. The previously recommended routes may contradict what we should do for the current queries from a global perspective. In other words, we would never make optimal decisions because we do not know future queries when processing the current one. For another, the route we return for each query causes conflicts under certain conditions. When app users drive along the route, they will occupy different road segments at different periods. Only when several vehicles, following their routes independently, utilize the same road segments at the same period do they increase the congestion of that road. We need to identify the congested road segments and periods and correctly measure the degree of congestion.

There is a large body of research on giving various alternative routes [5, 6, 13, 14, 35, 36, 39], in contrast to the fastest routes. However, their subjective opinions on a "good" route could be quite different and not specific to congestion minimization. Recent studies in the transportation area proposed many heuristics (*e.g.*, the entropy method [40]) for congestion minimization [19, 27, 40, 46, 52]. Nevertheless, they have no theoretical guarantee on the routing algorithms and are often unscalable to large road networks and online streaming queries.

Motivated by the above challenges, in this paper, we formulate a new problem called *Spatiotemporal Routing Problem* (SRP). It characterizes the online and temporal properties of routing queries on road networks and defines the congestion degree of a road segment at a period. Since we do not care about those uncrowded roads, we try to minimize the maximum congestion degree of the most congested road segment. Our solutions can respond to traffic changes quickly and recommend routes that improve the overall traffic status while incurring insignificant detour costs. To give some basic intuition about our challenges and solutions, we use the following toy example. We adopt the fastest route as a baseline.

Example 1. *In New York, a map platform receives 8 routing queries of the same type issued by 8 cars at some point, i.e., all from area A to B, shown in Figure 1a. This type of query has two options (corresponding to the two bridges) to reach area B: one is marked in blue with around 10-28 minutes and 5.9 km, and the other is marked in grey with around 12-35 minutes and 7 km. The baseline would return the first option for the queries since it is the fastest one. The bridge marked in red will be the most congested road segment with 8 cars. Our first routing algorithm, called Spatiotemporal Oblivious Routing (SOR), computes the degree of congestion that different routes would make on the future states of segments. It will guide 4 queries to use the one in grey, and the most congested segment (i.e., either of the bridges) now has 4 cars.*

Suppose that the platform next receives 8 queries of the other type from area C to D, shown in Figure 1b. This type of query has only



(a) First type of queries

(b) Second type of queries

Figure 1: An example of a road network and routing queries

one route option that traverses the congested bridge in red. We do not consider the route traversing the other bridge as an option because the detour is much longer than the fastest route. For the baseline, the most congested segment is the bridge in red with 16 cars. For SOR, since it also does not know the later 8 queries of the second type when processing those of the first type, its most congested segment is still the bridge in red with 12 cars. However, if we could gain some foresight of the future congested segments, we could have used the route in grey for all the 8 queries of the first type, making either of the bridges less crowded with 8 cars. Noticing some regular cases of traffic congestion in history, we propose the Spatiotemporal Routing with History (SRH) algorithm. By preprocessing the historical records, it will mark the bridge in red as a potential congested segment and avoid using it for the first 8 queries of the first type in online routing. It achieves the optimal solution of 8 cars in the most congested segment.

We summarize our contributions as follows.

- We give a formulation of the congestion minimization problem on road networks called the Spatiotemporal Routing Problem (SRP). It characterizes the congestion effect that each route causes in both spatial and temporal dimensions. We prove the NP-hardness of its offline setting.
- We propose the first routing algorithm called SOR with an $O(\ln n)$ theoretical guarantee in terms of the maximum number of vehicles compared with the optimal solution, where n is the number of nodes in the network.
- Based on SOR, we propose the SRH to use the history for further optimization. The theoretical guarantee is $O(\ln |C|)$, where C is a candidate set of potential congested segments, which is small and in the order of $10k$ in practice.
- We empirically demonstrate the effectiveness and efficiency of the proposed algorithms on real-world data. It could reduce the maximum number of vehicles of the baseline by 20% and handle a query in less than 20 ms.

The remainder of the paper is organized as follows. Section 2 defines the problem. Section 3 presents our two spatiotemporal routing algorithms. Section 4 gives some implementation details and techniques. Section 5 shows experimental results. Section 6 reviews the related work and Section 7 concludes our paper.

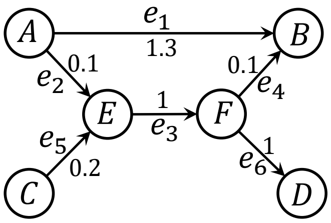


Figure 2: The road network of Example 1

2 PROBLEM STATEMENT

We first introduce several key concepts, then formally define our Spatiotemporal Routing Problem (SRP), and analyze its hardness.

2.1 Problem Definitions

Definition 1 (Road Network). *A road network $G(V, E, W)$ is represented by a connected directed graph, where V is the set of nodes and E is the set of edges. Let $n = |V|$ and $m = |E|$. Besides, for each edge $e \in E$, it is associated with a weight $w_e \in \mathbb{R}^+$ that represents the travel time on e, which may change over time.*

Definition 2 (Path). *A path p is a finite sequence of edges $p = (e_1, e_2, \ldots, e_k)$ such that the destination of e_i is the origin of e_{i+1} for $1 \leq i < k$. We define its weight (or travel time) $\gamma_p = \sum_{i=1}^{k} w_{e_i}$.*

Definition 3 (Fastest Path). *The fastest path from an origin node o to a destination node d is defined to be the one with the shortest travel time among all paths from o to d. Ties are broken arbitrarily.*

Since we focus on minimizing congestion, we simply assume an oracle to return the fastest paths. Note that state-of-the-art solutions could answer the fastest path queries efficiently [12, 25].

Definition 4 (Fastest Path Oracle). *The fastest path oracle efficiently responds to each fastest path query $q = (o_q, d_q)$, consisting of its origin o_q and destination d_q. We use $f^*(q)$ to denote the fastest path that the oracle returns for each query q.*

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

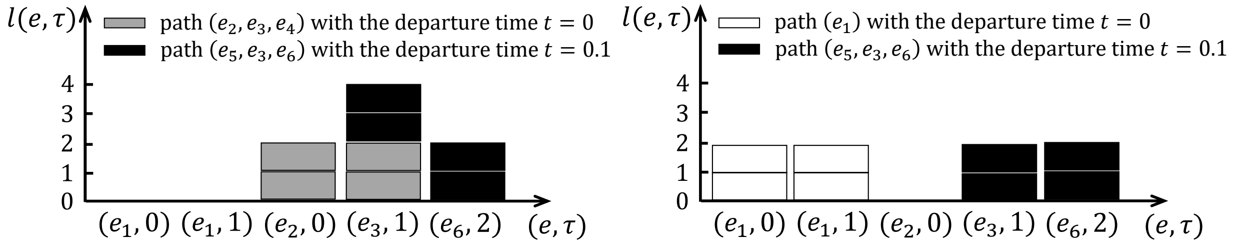
Definition 5 (Routing Query). *A routing query $q = (o_q, d_q)$ is issued at some time $t \in \mathbb{R}_{\geq 0}$ with its origin o_q and destination d_q. Since the drivers who issue routing queries often expect routes to have smaller travel time in general, we impose a detour constraint; that is, we should return a path p with its travel time*

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

where $a \geq 0$ is the detour factor used to control the detour cost. Besides, each query should be responded immediately.

We use $f(q)$ to denote any path satisfying the detour constraint.

Example 3. *Back to Example 2, suppose that there are two routing queries in the same form of $q = (A, B)$ issued at some time $t = 0$ and the detour factor $a = 0.1$. An alternative oracle could return the path $p = (e_1)$ with its travel time of 1.3, since the fastest path $f^*(q)$ has its travel time of 1.2 and the travel time of $p = (e_1)$ is smaller than $(1 + a)\gamma_{f^*(q)} = 1.32$, which satisfies the detour constraint.*



(a) The edge loads of the fastest path oracle

(b) The edge loads of the optimal solution

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

Definition 6 (Time Step). *We only monitor the traffic status at the time* $0, 1, \ldots,$ *called time steps, denoted by* $\tau \in \mathbb{N}$. *We simply assume that the status of these steps is representative and ignore the fluctuating status at any other time between steps.*

In reality, one minute per step is sufficient since this is also the frequency that Baidu Maps uses to update its traffic status [3].

On the time axis, time steps are integers and routing queries are issued at any time $t \in \mathbb{R}_{\geq 0}$ between steps. We simply assume that the departure time of any query to traverse its route p is the same as the issue time t (since we respond to each query within a negligible time compared to the length of each interval). Its arrival time is $t + \gamma_p \in \mathbb{R}_{\geq 0}$.

Definition 7 (Time Span). *The time span of a path p is defined as* $span(p) = \lceil \gamma_p \rceil$, *which is the smallest upper bound of the number of time steps for a vehicle to follow p. Let the maximum time span* $U = \max_p span(p)$ *for all paths.*

Example 4. *In Figure 2, the time span of the path* (e_5, e_3, e_6) *is* $\lceil 0.2 + 1 + 1 \rceil = 3$. *Since* (e_5, e_3, e_6) *is also the longest path in this network, the maximum time span* $U = 3$.

Note that U is at most 180 in all the experiments, corresponding to 3 hours for a trip in practice.

Definition 8 (Edge-step Pair). *Suppose that the last departure time of the paths that we return for queries received so far is* t'. *We examine the traffic status w.r.t. the edge-step pair* (e, τ) *for any* $e \in E$ *and any step* τ *in* $\{0, 1, \ldots, \lfloor t' \rfloor + U\}$.

Example 5. *Back to Figure 2, the edge-step pairs include* $(e_1, 0)$, $(e_1, 1), \ldots, (e_2, 0), (e_2, 1), \ldots, (e_6, 0), (e_6, 1), \ldots$.

The following location indicator is to specify whether a vehicle following its path will appear on some edge at some step.

Definition 9 (Location Indicator). *Given a path* $p = (e_1, \ldots, e_k)$ *with its departure time t, the location indicator* $\mathbb{1}_{p,t}(e, \tau)$ *for any edge-step pair* (e, τ) *is defined as 1 if and only if 1) e is some edge* e_i *in p and 2) the travel location lies 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* $i = 1$.

Since each path is usually associated with its departure time, we will use $\mathbb{1}_p(e, \tau)$ if the context is clear. We will sometimes say that the path p traverses the edge-step pair (e, τ) if $\mathbb{1}_p(e, \tau) = 1$.

Example 6. *Consider one path* $p = (e_5, e_3, e_6)$ *with its departure time* $t = 0.1$ *in Figure 2. The location indicators w.r.t. edge-step pairs* $(e_3, 1)$ *and* $(e_6, 2)$ *are equal to 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 where we check the traffic status.*

Definition 10 (Edge Load). *Given a set P of paths, the edge load* $l_P(e, \tau)$ *incurred by P on an edge-step pair* (e, τ) *is defined as the number of paths that traverse the edge-step pair* (e, τ), *i.e.,* $l_P(e, \tau) = \sum_{p \in P} \mathbb{1}_p(e, \tau)$.

Since edges may have different lengths, types, or the number of lanes, we may be interested in a normalized version of the edge load. We combine such factors by using the edge capacity, denoted by $cap(e)$. It is defined to be the maximum number of vehicles all traversing that edge at a given speed without delay. The normalized edge load is the number of paths which traverse the pair (e, τ) over $cap(e)$, *i.e.,* $l_P(e, \tau) = \sum_{p \in P} \mathbb{1}_p(e, \tau)/cap(e)$. For the simplicity of illustration, we omit the capacities since they are just normalized constants. More discussion can be found in our experiments where we consider the capacities. The later algorithms can be easily extended by replacing each edge load by its normalized version.

Example 7. *Figure 3a shows the edge loads (in the y-axis) of different edge-step pairs (in the x-axis) incurred by a path set P of four paths* $p_1 = p_2 = (e_2, e_3, e_4)$ *at* $t = 0$ *and* $p_3 = p_4 = (e_5, e_3, e_6)$ *at* $t = 0.1$. *Each block at one pair* (e, τ) *means that the location indicator is 1 for some path p with its departure time t,* i.e., $\mathbb{1}_{p,t}(e, \tau) = 1$. *For example, one grey block at* $(e_2, 0)$ *and one grey block at* $(e_3, 1)$ *correspond to the path* (e_2, e_3, e_4) *at* $t = 0$. *The edge load* $l_P(e_3, 1) = 4$ *since* $\mathbb{1}_p(e_3, 1) = 1$ *for all the four paths.*

Now, we are ready to define our Spatiotemporal Routing Problem (SRP) for congestion minimization.

Definition 11 (Spatiotemporal Routing Problem). *Given a network G, we receive online routing queries repeatedly and have to process them right away. We try to find an oracle f to return for each query q a path* $f(q)$ *while satisfying the detour constraint (Equation 1) and minimizing the maximum edge load incurred by the paths returned for all the queries Q received so far (i.e.,* $P = \{f(q) | q \in Q\}$*):*

$$\min_f \max_{e,\tau} l_{\{f(q)|q \in Q\}}(e, \tau). \tag{2}$$

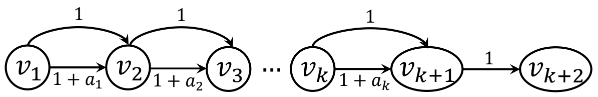


Figure 4: The network G in the NP-hard proof

Suppose that the last departure time of the paths is denoted by t'. The step τ is in $\{0, 1, \ldots, \lfloor t' \rfloor + U\}$, where U is the maximum time span. We may omit P and use $l(e, \tau)$ when we discuss all the paths so far.

Note that our problem can be applied to time scope of any length and networks of any small area in practice.

Example 8. *We use routing queries similar to those in Example 1 to illustrate the problem. Suppose that we first receive two queries $q_1 = q_2 = (A, B)$ all issued at time $t = 0$. The fastest path oracle will return the path (e_2, e_3, e_4) with the departure time $t = 0$ for the two queries since it has the shortest travel time of 1.2. The loads incurred by them are shown by the grey blocks in Figure 3a. Then, we receive the other two queries $q_3 = q_4 = (C, D)$ at time $t = 0.1$. We could only return the path (e_5, e_3, e_6) with the departure time $t = 0.1$, shown by the black blocks in Figure 3a. The maximum edge load for the fastest path oracle is thus $l(e_3, 1) = 4$ since $l(e_3, 1)$ is the largest one in Figure 3a. However, an optimal solution returns the path (e_1) for the first two queries, shown by the white blocks in Figure 3b. The maximum edge load of the optimal solution is thus equal to 2.*

2.2 Hardness

SRP is an online problem in the sense that when answering queries at any time, we know nothing about the future queries, which makes any algorithms suboptimal. The hypothetically optimal solution could be obtained by first collecting all the queries up to now without handling them and finding routes in hindsight. However, we will show that even if we know all the queries beforehand, also known as the offline version of this problem, it is still NP-hard.

Theorem 1. *The offline SRP is NP-hard.*

Proof. We will use the reduction from *the subset sum problem* [24], which is NP-complete, to the decision version of our SRP.

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 $b = x_1 a_1 + \cdots + x_k a_k$ holds. The decision version of our SRP asks whether the maximum edge load (Equation 2) could be at most some integer $L > 0$. In this proof, we will show that it is even NP-hard to find an answer with an integer $L = 1$.

Given an instance of the subset sum problem, we construct a graph G shown in Figure 4. The weights are shown beside the edges. There are exactly 2^k paths from the origin v_1 to the destination v_k; each top edge represents that we take x_i as 0 and each bottom one as 1. We will receive a query $q_1 = (v_1, v_{k+2})$ at time $t = 1$ and many queries of the other type constantly (v_{k+1}, v_{k+2}) for each step from $t = 1$ to T except for step $b + k + 1$. We could only return paths (v_{k+1}, v_{k+2}) for the routing queries of the latter type since there is only one path choice. For query q_1, it can be observed that if we route a path with its travel time exactly equal to $b + k$ (which means that it fills the gap of the queries of the latter type), the edge load is 1 for each step. Paths with lengths other than $b + k$ would result in a maximum edge load of 2 at one step. Hence, the subset sum problem has a yes answer if and only if the corresponding decision version of the SRP has a maximum load of 1. □

Table 1: Summary of notations

Notations	Descriptions
V, E	The node and edge sets of the network
$n = \lvert V \rvert, m = \lvert E \rvert$	The numbers of nodes and edges
W, w_e	The weight set and the edge weight
Q, q	The query set and a query
P, p	The path set and a path
$f^*(q), f(q)$	The fastest path and any other path
$span(p), U$	The time span of a path and the maximum one
$\mathbb{1}_{p,t}(e, \tau)$	The path p's location indicator for e and τ
$l_P(e, \tau)$	The edge load of (e, τ) incurred by paths P

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

Table 1 lists the main notations used throughout the paper.

3 ROUTING ALGORITHMS

This section introduces the routing algorithms SOR and SRH.

3.1 Oblivious Online Routing

When processing each new routing query q, we actually select a path from the set $\mathcal{P}(q) = \{p | \gamma_p \leq (1 + a)\gamma_{f^*(q)}\}$ where each path in it satisfies the detour constraint (Equation 1). To choose the best path $p \in \mathcal{P}(q)$ which causes less congestion, we need to give a metric to assess different paths. Specifically, since each path will traverse some edges at different steps, we first assign a variable $x(e, \tau)$ *w.r.t.* each edge-step pair (e, τ) to indicate the degree of congestion incurred by existing paths P. To evaluate a path $p \in \mathcal{P}(q)$, we use the sum of the variables *w.r.t.* those edge-step pairs that the path p traverses (*i.e.*, $\mathbb{1}_p(e, \tau) = 1$) as the metric value for the path p and choose the one with the minimum metric value.

A heuristic idea may directly make $x(e, \tau)$ equal to the edge load $l(e, \tau)$ (where P is omitted for simplicity) incurred by the paths so far. However, it could not differentiate the paths with the same sum of variables, as illustrated by the following example.

Example 9. *Suppose that the current edge loads $l(e_1, 0) = 4$, $l(e_1, 1) = 4$, $l(e_2, 0) = 1$, and $l(e_3, 1) = 7$. If we want to find a path for the query $q = (A, B)$ issued at time $t = 0$, one path (e_1) traverses the pairs $(e_1, 0)$ and $(e_1, 1)$, and the other path (e_2, e_3, e_4) traverses $(e_2, 0)$ and $(e_3, 1)$. The sum of variables for the two paths by directly using the edge loads are all equal to 8. However, we would prefer the path (e_1) because the maximum load after using (e_1) (which is $l(e_3, 1) = 7$ since $l(e_3, 1)$ is still the largest value after $l(e_1, 0)$ and $l(e_1, 1)$ are incremented by 1) is smaller than that of choosing (e_2, e_3, e_4) (which is 8 since $l(e_3, 1)$ is incremented by 1).*

Algorithm 1: Spatiotemporal Oblivious Routing

```
1  $x(e,\tau) \leftarrow \frac{1}{2Um}$ for all $(e,\tau)$ $(m = |E|)$
2  $l(e,\tau) \leftarrow 0$ for all $(e,\tau)$
3  $\Lambda \leftarrow 1$
4  foreach new query $q$ do
5      Let $\mathcal{P}(q)$ be the set of paths which satisfy the detour
         constraint (Equation 1)
6      Let $p \in \mathcal{P}(q)$ be the path with
         $\theta = \min_{p\in\mathcal{P}(q)} \sum_{e,\tau} \mathbb{1}_p(e,\tau)x(e,\tau)$
7      if $\theta > \Lambda$ or there is some $x(e,\tau) > \exp(\frac{1}{2})$ then
8          $\Lambda \leftarrow 2\Lambda$
9          $x(e,\tau) \leftarrow \frac{(1+\frac{1}{2\Lambda})^{l(e,\tau)}}{2Um}$ for all $e,\tau$
10         Let $p \in \mathcal{P}(q)$ be the path with
             $\theta = \min_{p\in\mathcal{P}(q)} \sum_{e,\tau} \mathbb{1}_p(e,\tau)x(e,\tau)$
11     route the path $p$ for query $q$, i.e., $f(q) = p$
12     foreach $e,\tau$ such that $\mathbb{1}_{f(q)}(e,\tau) = 1$ do
13         $x(e,\tau) \leftarrow (1+\frac{1}{2\Lambda})x(e,\tau)$
14         $l(e,\tau) \leftarrow l(e,\tau)+1$
```

Besides, directly using the edge load builds no connection with the optimal solution, thus providing no guarantee on the maximum edge load. We would like to use an exponential growth function for these variables (which overcomes the first weakness since we will choose the path (e_1) in Example 9 because $b^4 + b^4 < b^7 + b$ for any $b > 0$ and $b \neq 1$ by the inequality of arithmetic and geometric means) and a different way of updating them (which involves an estimate of the optimal solution). Specifically, let

$$x(e,\tau) = \frac{(1+\frac{1}{2\Lambda})^{l(e,\tau)}}{2Um}, \tag{3}$$

where U is the maximum time span and $\Lambda > 0$ is an estimate of the optimal solution (detailed later). SOR mainly finds the path $p \in \mathcal{P}(q)$ with the minimum sum of the variables.

Algorithm 1 summarizes the whole procedure. In lines 1-3, we initialize each variable $x(e,\tau)$ as $\frac{1}{2Um}$ (since each edge load $l(e,\tau)$ is initially 0) and the initial estimate of the optimal solution Λ as 1. Note that how to incorporate the existing traffic in the initialization is discussed in Section 4.2. 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 5-6. In lines 11-14, after choosing the path $f(q)$ for the new query, we update those variables regarding the edge-step pairs that the path $f(q)$ traverses (*i.e.*, $\mathbb{1}_{f(q)}(e,\tau) = 1$) by multiplying $(1+\frac{1}{2\Lambda})$ in line 13 (since the edge loads are increased by 1). We update the edge loads in line 14. In lines 7-10, we mainly update the estimate of the optimal solution Λ. In line 7, the conditions $\theta \geq \Lambda$ and $x(e,\tau) \geq \exp(\frac{1}{2})$ both indicate that the estimate may be smaller (proved later). Intuitively, the two conditions mean that the sum of variables or any single variable is large enough and updated many times. Thus, the estimate of the optimal solution should also be increased. We simply double the estimate Λ in line 8. We also reset all the variables in line 9 and find the path again in line 10, since the multiplier $(1+\frac{1}{2\Lambda})$ changes when the estimate Λ doubles.

Note that the time-consuming part lies in lines 5-6 where we need to find the path. We will use a Dijkstra-based algorithm with a pruning technique discussed in Section 4.1. Its time complexity in the worst case is $O(m \ln n)$.

Example 10. *Back to Example 8, the maximum time span $U = 3$ since the longest path (e_5, e_3, e_6) has a time span equal to $\lceil 2.2 \rceil = 3$. Each variable $x(e,\tau) = \frac{1}{2Um} = \frac{1}{36}$ at first since there are 6 edges. For the first query $q_1 = (A, B)$ at time $t = 0$, 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 choose the path (e_1). The two variables $x(e_1, 0)$ and $x(e_1, 1)$ are all updated as $(1 + \frac{1}{2\Lambda})\frac{1}{36} = \frac{1}{24}$. For the query $q_2 = (A, B)$, Algorithm 1 will definitely choose 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 two queries q_3, q_4 with (C, D), it has only one choice (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.*

Lemma 2 gives us some intuition about why we initialize and update the variable $x(e,\tau)$ as in Equation 3. Suppose that the last departure time of the paths returned so far is t' and let $\tau' = \lfloor t' \rfloor$.

Lemma 2. *The edge load for each edge-step pair (e,τ) incurred by all the paths so far is $O(\Lambda \ln n)$, in terms of the final estimate of the optimal solution Λ.*

Proof. We initialize each variable $x(e,\tau)$ as $\frac{1}{2Um}$ and increase it to at most $(1 + \frac{1}{2\Lambda})\exp(\frac{1}{2}) \leq \frac{3}{2}\exp(\frac{1}{2})$ (since $x(e,\tau) \leq \exp(\frac{1}{2})$ before we update Λ and $\Lambda \geq 1$). We have $\frac{(1+\frac{1}{2\Lambda})^{l(e,\tau)}}{2Um} \leq \frac{3}{2}\exp(\frac{1}{2})$. After rearranging the inequality, we obtain $l(e,\tau) = O(\Lambda \ln n)$. □

We will show next that $\Lambda/2 < OPT$, leading to the $O(\ln n)$ competitive ratio, since any edge load $l(e,\tau) = O(\ln n) \cdot OPT$. Let $OPT_{[0,\tau']}$ and $OPT_{[\tau'+1,\tau'+U]}$ be the maximum edge load of the optimal solution from step 0 to τ' and step $\tau'+1$ to $\tau'+U$, respectively. It is easy to see that $OPT = \max(OPT_{[0,\tau']}, OPT_{[\tau'+1,\tau'+U]})$. To show $\Lambda/2 < OPT$, we first prove Lemma 3.

Lemma 3. *When $\theta > \Lambda$ or $x(e,\tau) > \exp(\frac{1}{2})$ for some e and τ, the current estimate of the optimal solution Λ is smaller than $OPT_{[\tau',\tau'+U]}$ in terms of the last departure time t' of the path before the current iteration and $\tau' = \lfloor t' \rfloor$.*

Proof. The idea of using the primal and dual programs is based on [11]. We construct the following linear program where $y(p)$ is the variable indicating whether we choose the path p in the solution. The first set of constraints states that we could only choose one path $p \in \mathcal{P}(q)$ for a query q. The second set of constraints state that for any e and τ such that $\tau' + 1 \leq \tau \leq \tau' + U$, its edge load should be smaller than Λ. This is to ensure that if $\Lambda > OPT_{[\tau'+1,\tau'+U]}$, we could construct the optimal solution to this program with its objective value equal to $|Q|$, where Q denotes the queries up to t'. In other words, if we assign each variable $y(p)$ as the optimal solution, we could answer all the queries while making any edge load on step $\tau' + 1$ to $\tau' + U$ smaller than $OPT_{[\tau'+1,\tau'+U]} \leq \Lambda$.

$$\begin{array}{lll} \max & \sum_{q\in Q}\sum_{p\in\mathcal{P}(q)} y(p) & \\ s.t. & \sum_{p\in\mathcal{P}(q)} y(p) \leq 1, & \forall q\in Q \\ & \sum_{p:\mathbb{1}_{p,t}(e,\tau)=1} y(p)/\Lambda \leq 1, & \forall e\in E, \tau: \tau'+1\leq\tau\leq\tau'+U. \end{array}$$

Its dual program (shown below) and the primal one are used to check if it has a feasible solution with the objective value $|Q|$. If it does not, it could only be because $OPT_{[\tau'+1,\tau'+U]} > \Lambda$.

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

Let $\Gamma = \theta/\Lambda$. Each time we set a variable $x(e,\tau)$, we always set a variables $z(q)$ so that the constraints are met with equality. After routing the path p in line 12, we set $z(q)$ as $1-\Gamma$. Each time we route a path, the dual value increases at most $\Gamma/2+1-\Gamma = 1-\Gamma/2$ (where the first $\Gamma/2$ is derived by expanding the formula in line 13).

For the last query q, if we find $\Gamma > 1$ for some path p, the dual solution has already been feasible; that is, all the constraints for potential paths related to this query are satisfied, since taking the minimum sum of variables as the first term can make the constraints 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 $1-\Gamma/2$ and the sum of variables is $\frac{1}{2}$ initially). But when $\Lambda > OPT_{[\tau'+1,\tau'+U]}$, we must have a primal objective value $|Q|$, contradicting the dual solution with a smaller value.

When $x(e,\tau) > \exp(\frac{1}{2})$, we only consider those iterations where $x(e,\tau)$ is increased from 1 to $\exp(\frac{1}{2})$ and use 1 as the upper bound for other iterations. There are at least Λ iterations for $x(e,\tau)$ to be greater than $\exp(\frac{1}{2})$ since $(1+\frac{1}{2\Lambda})^{\Lambda} \leq \exp(\frac{1}{2})$. When $x(e,\tau)\geq 1$, we also have $\Gamma \geq x(e,\tau)/\Lambda \geq 1/\Lambda$. We set $z(q) = 1$ for the last request to make a feasible dual solution with its objective value strictly smaller than $\frac{1}{2}+|Q|-1-\Lambda+\Lambda(1-\frac{1}{2\Lambda})+1 = |Q|$, which contradicts the fact as before. □

Theorem 4. *Algorithm 1 is $O(\ln n)$ competitive.*

Proof. We need to prove that in terms of the final Λ, $\Lambda/2 < OPT = \max(OPT_{[0,\tau']}, OPT_{[\tau'+1,\tau'+U]})$. If $OPT = OPT_{[\tau'+1,\tau'+U]}$, $\Lambda/2 < OPT_{[\tau'+1,\tau'+U]} = OPT$ by Lemma 3 which states that Λ should be smaller than $OPT_{[\tau'+1,\tau'+U]}$ before the last time we double it. If $OPT = OPT_{[0,\tau']}$, the statement holds because $\Lambda/2 < OPT_{[\tau'+1,\tau'+U]} \leq OPT_{[0,\tau']}$. Combining Lemma 2, the maximum edge load is at most $O(\Lambda\ln n) = O(2\ln n)\cdot OPT$. □

3.2 Online Routing with History

The previous algorithm assumes that we know nothing about the future traffic condition. However, if we know that some road segments are about to be crowded, we could avoid choosing the paths that will traverse them. Since real traffic is often periodic (*e.g.*, in a cycle of 24 hours), we could obtain some 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 choosing 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 propose three techniques for generating a candidate set of edge-step pairs (denoted by C) which are to be congested with high probability and an algorithm called SRH with a competitive ratio of $O(\ln(|C|))$.

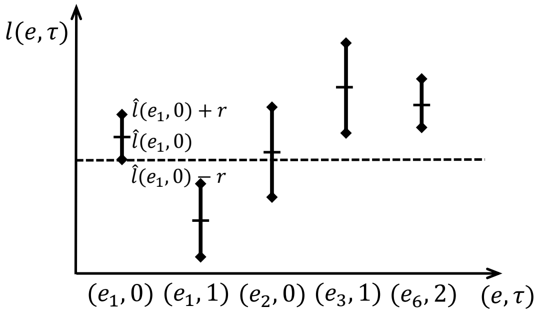


Figure 5: The confidence bounds of edge loads

3.2.1 Estimating the Edge Load. In the following, we would 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 we know that the true edge load of an edge-step pair (e,τ) is smaller than some other edge load of a pair (e',τ') with high probability, we could prune the pair (e,τ) since we only care about the maximum edge load. Then, we would prune all the edge-step pairs for which we can find such pairs with higher loads stated above. To find such events of high probability, we first obtain a confidence interval for each edge load $l(e,\tau)$, defined by *lower* and *upper confidence bounds* (detailed in Equation 4), and each edge load $l(e,\tau)$ lies in its confidence 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. For example, 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 of $(e_1,0)$ is shown by the dashed line with its y-axis value of $\widehat{l}(e_1,0)-r$. Since it is greater than the upper bound of the load of $(e_1,1)$, we can safely prune $(e_1,1)$. Note that r is called the *confidence radius.*

We first give the lower and upper bounds of $l(e,\tau)$. By the Hoeffding bound [31], 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 edge loads corresponding to the pair (e,τ) in cycles. Using an equivalent form, we have the following with probability $1-\delta/2$,

$$l(e,\tau) \in [\widehat{l}(e,\tau)-r(e,\tau), \widehat{l}(e,\tau)+r(e,\tau)], \tag{4}$$

where the confidence radius $r(e,\tau) = \sqrt{\frac{R^2(\ln 4-\ln\delta)}{2N}}$. This can be proved by replacing the constant ϵ with the confidence radius in the Hoeffding bound.

To prune all the pairs which have the other pair (e',τ') with a greater edge load, we first sort all the edge-step pairs in the descending order of upper bounds $\widehat{l}(e,\tau)+r(e,\tau)$. When checking

Algorithm 2: Pruning Low Edge Loads

```
input  : The statistics $\widehat{l}(e,\tau)$ and $r(e,\tau)$ from historical
         records
output : A candidate set $C$
1 Sort all $\{(e,\tau)|e \in E, \tau = 0,1,\dots,T\}$ in the descending order
  of $\widehat{l}(e,\tau) + r(e,\tau)$
2 $C \leftarrow \emptyset$
3 foreach $(e,\tau)$ in the sorted list do
4     $lb \leftarrow \max(lb, \widehat{l}(e,\tau) - r(e,\tau))$
5     if $\widehat{l}(e,\tau) + r(e,\tau) < lb$ then
6         break
7     $C \leftarrow C \cup \{(e,\tau)\}$
8 return $C$
```

each pair in this order, if the current upper bound has been smaller than the lower bound of the load of a pair, we do not have to check the rest. Note that we should also maintain a maximal lower bound when iterating the pairs.

The pruning procedure is summarized in Algorithm 2. In line 1, we sort all the edge-step pairs in the descending order of 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 5. We initialize the candidate set C as an empty set in line 2 and update it in line 7.

Theorem 5. *With probability $1-\delta$, any pair (e,τ) whose $e \in E$, $\tau \in \{0,1,\dots,T\}$, and $\widehat{l}(e,\tau) + r(e,\tau) < lb$ 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 $1-\delta$. □

3.2.2 *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^t denote the set of queries from step t to $t+1$. We consider the estimate $|\widehat{Q^t}|$ at step t.

The basic intuition is as follows. At each step t, we maintain the current maximum edge load up to step t, denoted by $obj(t)$. Before processing the query set Q^t whose queries appear between step t and $t+1$, we ignore those edge-step pairs at step t such that their edge loads are so small that they cannot be the maximum one at step $t+1$; formally, $l(e,t+1)$ cannot be the maximum load at step $t+1$ if the load is smaller than the current maximum edge load $obj(t)$ even when all the paths that return for Q^t traverse $(e,t+1)$, *i.e.*, $l(e,t) + |Q^t| \le obj(t)$. Finally, we just need to use the upper confidence bound of $|Q^t|$ in the pruning condition as in the previous technique. Note that this is a procedure that has to be done at each step t.

Algorithm 3: Selecting Representative Edge Loads

```
input  : The candidate set $C$, statistics $\widehat{l}(e,\tau)$ and
         $r(e_1,\tau_1,e_2,\tau_2)$ from historical records, and a
         threshold $\beta \ge 0$
output : A candidate set $C$
for $\tau = 1,2,\dots$ do
    foreach $v \in V$ do
        for any pairs of $e_1$ and $e_2$ incident to $v$ do
            if $\widehat{l}(e_1,\tau) - \widehat{l}(e_2,\tau) + r(e_1,\tau,e_2,\tau) < \beta$ where
              $(e_1,\tau) \in C$ then
                $C \leftarrow C \cup \{(e_2,\tau)\}$;
foreach $e \in E$ do
    for any pairs of $\tau_1, \tau_2$ where $|\tau_1 - \tau_2| = 1$ do
        if $\widehat{l}(e,\tau_1) - \widehat{l}(e,\tau_2) + r(e,\tau_1,e,\tau_2) < \beta$ where
          $(e,\tau_1) \in C$ then
            $C \leftarrow C \cup \{(e,\tau_2)\}$;
return $C$
```

The confidence radius of $|Q^t|$ is denoted by $r(Q^t)$, with the same form as in $l(e,\tau)$. The theorem below is used in lines 5-9 of Algorithm 4.

Theorem 6. *Let Q^t be the set of queries from step t to $t+1$. With probability $1-\delta$, any pair $(e,t+1)$ whose $e \in E$ and $l(e,t) + |\widehat{Q^t}| + r(Q^t) \le obj(t)$ is not the one with the maximum load at step $t+1$.*

Proof. By the Hoeffding bound, we have $Q^t \le |\widehat{Q^t}| + r(Q^t)$ with probability $1-\delta$. Hence, $l(e,t) + Q^t \le obj(t)$ holds with probability $1-\delta$, and the pair $(e,t+1)$ could not achieve the maximum load at step $t+1$ even if all the paths routed for queries Q^t traverse $(e,t+1)$. □

3.2.3 *Finding Spatial and Temporal Correlations.*

In real traffic, there is often a correlation among the edge loads of some adjacent road segments at the same step or some consecutive steps of one road segment. If one pair $(e,\tau) \in C$, then any pair that has a strong correlation with it should also be included in C. Specifically, we would like to find some pairs (e_1,τ_1) and (e_2,τ_2) such that $|l(e_1,\tau_1) - l(e_2,\tau_2)| \le \epsilon$ with high probability, where ϵ is a constant representing the gap that we want to control.

We first give the probability bounds by Chebyshev's inequality:

$$\mathbb{P}(|l(e_1,\tau_1)-l(e_2,\tau_2)-\widehat{l}(e_1,\tau_1)+\widehat{l}(e_2,\tau_2)| \ge \epsilon\sigma_{l(e_1,\tau_1)-l(e_2,\tau_2)}) \le \frac{1}{\epsilon^2},$$

where $\sigma_{l(e_1,\tau_1)-l(e_2,\tau_2)}$ is the variance of the random variable $l(e_1,\tau_1) - l(e_2,\tau_2)$. It can be derived by

$$\sigma_{l(e_1,\tau_1)-l(e_2,\tau_2)} = \sqrt{\sigma^2_{l(e_1,\tau_1)} + \sigma^2_{l(e_2,\tau_2)} - 2cov(l(e_1,\tau_1), l(e_2,\tau_2))}.$$

For any random variable X with N samples, $x_1, x_2, \dots$, the standard deviation σ_X is simply estimated by $\sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$, where $\overline{x}$ is the sample mean, and the covariance $cov(X,Y)$ by $\frac{\sum_{i=1}^{N}(x_i-\bar{x})(y_i-\bar{y})}{N-1}$. If we want to make the probability lower than some δ, we could

```
Algorithm 4: Spatiotemporal Routing with History
1  $x(e,\tau) \leftarrow 1/(2|C|)$ for all $(e,\tau)$
2  $l(e,\tau) \leftarrow 0$ for all $(e,\tau)$
3  $\Lambda \leftarrow 1$
4  for time step $t = 0, 1, \dots$ do
5      $obj(t) = \max_e l(e,t)$
6      $C' \leftarrow C$
7      foreach $e \in E$ do
8          if $l(e,t) + |\widehat{Q^t}| + r(Q^t) \le obj(t)$ then
9              $C' \leftarrow C' \setminus \{(e, t+1)\}$
10     foreach new query $q \in Q^t$ do
11         Let $\mathcal{P}(q)$ be the set of paths which satisfy the detour
           constraint (Equation 1)
12         Let $p \in \mathcal{P}(q)$ be the path with
           $\theta = \min_{p \in \mathcal{P}(q)} \sum_{(e,\tau) \in C'} \mathbb{1}_p(e,\tau) x(e,\tau)$
13         if $\theta > \Lambda$ or there is some $x(e,\tau) > \exp(\frac{1}{2})$ then
14             $\Lambda \leftarrow 2\Lambda$
15             $x(e,\tau) \leftarrow \frac{(1+\frac{1}{2\Lambda})^{l(e,\tau)}}{2|C|}$ for all $e, \tau$
16             Let $p \in \mathcal{P}(q)$ be the path with
               $\theta = \min_{p \in \mathcal{P}(q)} \sum_{(e,\tau) \in C'} \mathbb{1}_p(e,\tau) x(e,\tau)$
17         route the path $p$ for query $q$, i.e., $f(q) = p$
18         foreach $e, \tau$ such that $\mathbb{1}_{f(q)}(e,\tau) = 1$ do
19             $x(e,\tau) \leftarrow (1 + \frac{1}{2\Lambda}) x(e,\tau)$
20             $l(e,\tau) \leftarrow l(e,\tau) + 1$
```

obtain a similar confidence radius $r(e_1, \tau_1, e_2, \tau_2)$ for the random variable $l(e_1, \tau_1) - l(e_2, \tau_2)$ as in Section 3.2.1. We will include the pair in C if the other one is in C when the upper confidence bound of the difference $\widehat{l}(e_1, \tau) - \widehat{l}(e_2, \tau) + r(e_1, \tau, e_2, \tau)$ is smaller than a threshold $\beta \ge 0$.

The procedure is illustrated in Algorithm 3. We consider the edges incident to the same node v in lines 1-7 and consecutive time steps of the same edge in lines 8 – 14. In lines 4-5 and 8-9, we include the other pair if it has a strong correlation with some pair in the candidate set, *i.e.*, the upper confidence bound of their difference is smaller than a threshold β.

3.2.4 Optimized Online Routing. All the techniques above are summarized in Algorithm 4. For simplicity of the pseudo-code, let Q^t be the set of queries shown from step t to $t+1$ which could still appear at any time between step t and $t+1$. The basic procedure is similar to Algorithm 1. We can imagine that there are Um candidate pairs in Algorithm 1 (*i.e.*, $C = \{(e,\tau) | e \in E, \tau = t+1, \dots, t+U\}$) and it chooses those whose location indicators are equal to 1. Similarly, Algorithm 4 uses the edge-step pairs in a different C whose location indicators are equal to 1. The only three differences are that we first generate C by Algorithm 2 and Algorithm 3, that we apply the technique in Section 3.2.2 in lines 5-9, and that we replace Um by $|C|$ in lines 1 and 15 and the sum of variables in lines 12 and 16.

Note that the time complexity is the same as Algorithm 1 which is $O(m \ln n)$.

Example 11. *Suppose that $C = \{(e_3, 1)\}$. All the variables are initialized as $\frac{1}{2|C|} = \frac{1}{2}$. Algorithm 4 would choose the path (e_1) for the first two queries q_1 and q_2 with (A, B). This is because the path (e_1) contains no pair in C and has a sum of variables equal to 0, but that 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 and q_4 with (C, D), it has no choice but (e_5, e_3, e_6). The variable $x(e_3, 1)$ is first updated as $(1 + \frac{1}{\Lambda})\frac{1}{2} = 3/4$ and then $(1 + \frac{1}{\Lambda})\frac{3}{4} = 9/8$ after the query q_4. Algorithm 4 achieves a maximum edge load $l(e_3, 1) = 2$, equal to the optimal solution.*

Theorem 7. *Algorithm 4 achieves a competitive ratio of $O(\ln |C|)$ with probability $1 - \delta$.*

Proof. We construct the following programs.

$$\begin{aligned} \max \quad & \sum_{i=1}^{t} \sum_{q \in Q^i} \sum_{p \in \mathcal{P}(q)} y(p) \\ s.t. \quad & \textstyle\sum_{p \in \mathcal{P}(q)} y(p) \le 1, \quad \forall q \in \cup_{i=1}^{t} Q^i \\ & \textstyle\sum_{p: \mathbb{1}_{p,t}(e,\tau)=1} y(p)/\Lambda \le 1, \quad \forall (e,\tau) \in C. \end{aligned}$$

$$\begin{aligned} \min \quad & \sum_{(e,\tau) \in C} x(e,\tau) + \sum_{q} z(q) \\ s.t. \quad & \textstyle\sum_{(e,\tau) \in C: \mathbb{1}_{p,t}(e,\tau)=1} x(e.\tau)/\Lambda + z(q) \ge 1, \quad \forall q, p \in \mathcal{P}(q). \end{aligned}$$

Since we prune the candidate pairs correctly with probability $1-\delta$ as discussed above, we will regard those events as deterministic ones in the following. We construct the first program with constraints only for those $(e,\tau) \in C$ and the second one with the sum of terms *w.r.t.* $(e,\tau) \in C$ in the constraints. The initial primal objective value starts with 1/2. We get a similar result to that in the proof of Lemma 2: $(1 + \frac{1}{2\Lambda})^{l(e,\tau)} \frac{1}{2|C|} \le \frac{3}{2}\exp(\frac{1}{2})$. By rearranging the inequality, we have $l(e,\tau) = O(\ln(|C|)) \cdot \Lambda$. Since the maximum edge load will occur in C with high probability, we still have $\Lambda/2 < OPT$ and $l(e,\tau) = O(\ln(|C|)) \cdot OPT$ □

4 PRACTICAL IMPLEMENTATION

In this section, we give some details about how to implement the two algorithms in practice.

4.1 Candidate Paths

The algorithms need to find for each query q the set of paths $\mathcal{P}(q)$ satisfying the detour constraint and the one with the minimum $\sum_{(e,\tau)} \mathbb{1}_{p,t}(e,\tau) x(e,\tau)$. To implement this, we design a Dijkstra-based algorithm with a pruning technique. We still follow the Dijkstra procedure [21]. However, since we want to make sure that the resulting path satisfies the detour constraint, we have to prune some paths when visiting a node. Suppose that we are about to insert the neighbors of the current node into the priority queue. The pruning strategy is that if the travel time from the origin to the neighbor node plus the shortest travel time from the neighbor node to the destination cannot satisfy the detour constraint, we prune this neighbor node since other paths must be longer. This pruning further requires us to compute the shortest travel time from those nodes to the destination, which can be done by a backward Dijkstra's search from the destination to the origin for each query. To utilize the Dijkstra framework to find the minimum value, we

Table 2: Real dataset statistics

Datasets	$|Q|$	Time Periods
NYC	20775	5pm-6pm
Baidu Maps	22881	7am-7:30am
Didi	22794	9am-9:35am

partition the sum of variables into different parts *w.r.t.* the edges. When relaxing neighbor edges of a node, since we know the travel time from the origin to this node, we could know the starting step at which we begin to use this edge and only use the sum of variables corresponding to it. This can be seen as reassigning static edge weights for edges. However, they are dynamic because we could see an edge with different degrees of congestion when arriving at the origin of this edge at a different time. This way helps us find the minimum sum of variables as long as we assume that no one will wait at some node. A more rigorous way of generating the path is to allow visiting each node several times. Also note that state-of-the-art efficient solutions for computing shortest paths could be easily applied as long as we follow the pruning strategy above. Moreover, we focus on minimizing congestion and show in the experiments that even the algorithms based on the Dijkstra procedure are very fast per query and sufficient for road networks of modern cities.

4.2 Real Traffic

At each step, there have been many existing external vehicles running on the roads. The platforms may not know their complete trajectories but only the current number of vehicles on each road segment. If we want to take the current traffic status into account and improve the future status, we could reflect those existing vehicles in each variable $x(e, \tau)$ (for all edges and only the current step). The way is similar as we reset the variables. Specifically, at step τ, if there are currently k vehicles on the edge e, we additionally multiply the variable $x(e, \tau)$ by $(1 + \frac{1}{2\Lambda})^k$ in terms of the current estimate of the optimal solution Λ at step τ. Similar to what we have discussed in Section 2, when we consider different edge capacities, denoted by $cap(e)$, we additionally divide the number of vehicles k by the corresponding edge capacity $cap(e)$. Note that we have to update $x(e, \tau)$ for every step according to the traffic status since we do not have their complete trajectories.

5 EXPERIMENTAL STUDY

This section shows the performance of the two proposed solutions on three real-world datasets.

5.1 Experiment Setup

In our experiments, algorithms were implemented by the compiler g++ 4.9.2 and performed in a machine with 2.66GHz CPU and 48GB RAM installed with CentOS 5 Linux distribution.

Datasets. We collected road network and query data. We use two road networks from New York City (NYC) and Beijing (BJ). The NYC network from DIMACS[1] has 264,346 nodes and 733,846 edges, and the Beijing network from OpenStreetMap[2] has 188,229 nodes and 436,648 edges. For queries, we use NYC taxi trips in August 2013 from NYC TLC Trip Record Data[3], routing queries in April 2017 from Baidu Maps's Q-Traffic Dataset [38], and online taxi-calling trips in October 2016 from Didi[4]. The NYC queries were tested on the NYC road network, and the queries from Baidu Maps and Didi were tested on the Beijing road network. Since we have the longitude and latitude of origins and destinations of queries, we map them into the network nodes by building a k-d tree on network nodes and finding their nearest neighbors. We define the interim of time steps as one minute since the congestion status is updated every one minute in Baidu Maps [3].

We evaluated our algorithms by varying the query set Q and the detour factor a. Following existing work [45, 56, 59], we use their standard by setting $|Q| = [5000, 7500, \underline{10000}, 12500, 15000]$, where the default value is underlined. We extracted the query sets of the above sizes from three sources, as shown in Table 2. We also conducted the scalability test by varying $|Q| = [1, 2, 3, 4, 5] \times 10^5$ from NYC data. The original NYC queries are repeated many times to meet the corresponding $|Q|$. For the detour factor a, since a small value could result in less detour cost, we focus on small $a = [\underline{0.05}, 0.075, 0.1, 0.125, 0.15]$ from the three sources in Table 2 but also test large values from 0.1 to 0.5. Note that SRH uses a set C of candidates from history to guide its routing. For each of the three sources, we generate and use only one set C of candidate pairs by considering the corresponding time periods in the previous 20 days. The algorithms use the same parameters $\delta = 0.1$ and $\beta = 2$.

Compared algorithms. We compared three related algorithms from three branches: shortest paths, alternative paths, and congestion avoidance strategies.

(1) ShortestPath. Using the shortest paths is one solution for processing routing queries [45, 54, 56, 59]. We implement it using a Dijkstra-based idea and count the corresponding edge loads.

(2) Iterative Penalty Method (IPM). The representative approach of alternative paths is the IPM which gives alternative paths by imposing a penalty weight on the traversed edges [6]. Since the recent study shows that a penalty weight of 1.4 performs the best in practice [36], we implement this variant for comparison.

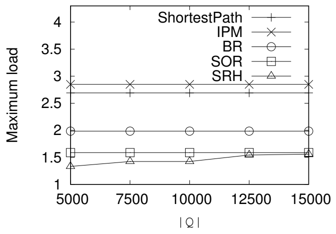
(3) Balanced Routing (BR). BR is the state-of-the-art congestion avoidance strategy in the transportation area [27]. It uses the entropy metric to choose the best path from a set of dissimilar paths [40].

(4) SOR. Our first algorithm uses no history and evaluates different routes based on the variables of edge-step pairs indicating the degree of congestion.

(5) SRH. Our second improvement uses the history to generate a set of candidate pairs to enlighten the congested edges and time steps and then guide the routing by the candidate pairs.

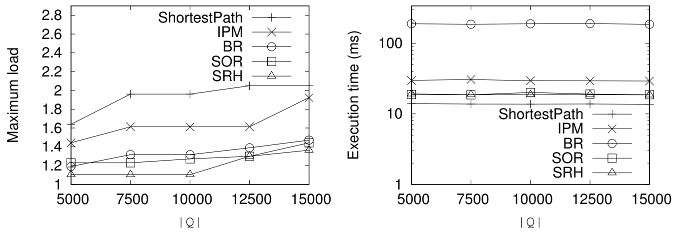
We assess the performance of the above algorithms in terms of the maximum edge load and the average execution time. Each edge load is normalized by the edge capacity of each edge, which is the maximum possible number of vehicles traversing that edge at a given normal speed without delay. Each edge capacity is calculated based on the number of lanes, the length of the edge, and the average spacing between vehicles without delay [2]. Basically, the edge load

[1] http://www.dis.uniroma1.it/challenge9/download.shtml
[2] https://download.bbbike.org/osm/bbbike/Beijing/
[3] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[4] https://outreach.didichuxing.com/research/opendata/



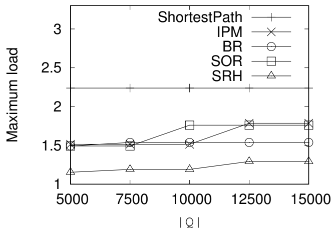
(a) **Max. load of varying $|Q|$ (NYC)** (b) **Time of varying $|Q|$ (NYC)**

Figure 6: Varying $|Q|$ on NYC



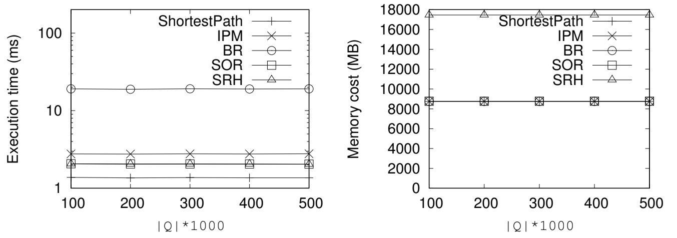
(a) **Max. load of varying $|Q|$ (Baidu)** (b) **Time of varying $|Q|$ (Baidu)**

Figure 7: Varying $|Q|$ on Baidu



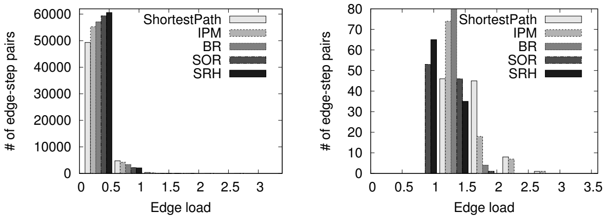
(a) **Max. load of varying $|Q|$ (Didi)** (b) **Time of varying $|Q|$ (Didi)**

Figure 8: Varying $|Q|$ on Didi



(a) **Max. load of varying $|Q|$ (NYC)** (b) **Time of varying $|Q|$ (NYC)**

Figure 9: Scalability test



(a) **All edge-step pairs** (b) **Top 100 edge-step pairs**

Figure 10: Results of load distribution on NYC data

of an edge-step pair less than one means that there is no congestion and all vehicles can traverse that edge at a given normal speed at the time of the corresponding step. However, the maximum edge load greater than one indicates a certain degree of congestion.

5.2 Experiment Results

5.2.1 Effect of $|Q|$. Figure 6, 7, and 8 show the results of varying the number of queries $|Q|$ on the NYC, Baidu, and Didi datasets, respectively.

For the maximum load, all the algorithms have larger values when $|Q|$ is greater. This is because more paths occupy more edge-step pairs within the whole limited network. The maximum load of all algorithms is larger than one, indicating that the congestion does appear for some edge-step pairs. Among all the algorithms, *ShortestPath* performs the worst since it does not consider the congestion status of the edges. Different paths could easily conflict with each other on some edge-step pairs. *IPM* achieves greater maximum loads on NYC and Baidu datasets but smaller ones on Didi datasets. *BR* is basically at the medium level and *SOR* is the second best one on all datasets. *SRH* always achieves the smallest maximum load among all the algorithms. It could reduce the maximum load of the baseline by nearly 20% on Didi datasets.

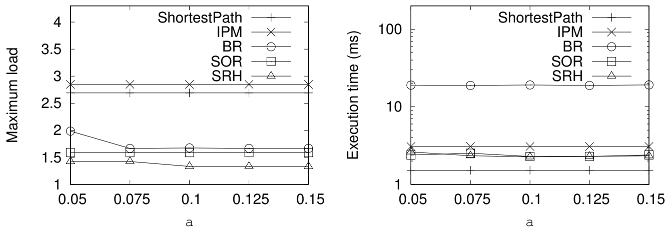
For the execution time, all the algorithms take a constant time per query when $|Q|$ is greater. *BR* runs slowly on all datasets because it needs to compute and compare multiple paths for every single query. The time cost of the rest of the algorithms is around 10 ms and hence acceptable in practice, but *BR* is infeasible in a real-world application. Note that though *ShortestPath* is the fastest one for giving paths, our proposed algorithms optimize the congestion status just a bit slower.

5.2.2 Load distribution. Figure 10 shows the load distribution on the default NYC dataset. We split edge-step pairs into equal-width intervals according to their edge loads and count the number of edge-step pairs for each interval. Note that we omit those edge-step pairs with zero edge loads.

Figure 10a shows the load distribution of all edge-step pairs. It can be seen that most pairs have their edge loads less than 0.5. We need to focus on the tiny minority of pairs with loads greater than 1 because those most congested ones are the bottleneck for congestion minimization.

Since it is hard to observe the distribution for edge loads greater than 1, we only consider the top 100 edge-step pairs with the greatest edge loads in Figure 10b. We can find that the top 100 pairs basically have edge loads greater than 1. The pairs of *SOR* and *SRH*, shown by dark black bars, concentrate on the left interval [1,1.5]. However, *ShortestPath* and *IPM* could have edge loads greater than 2. The results are consistent with the least maximum load of *SRH*.

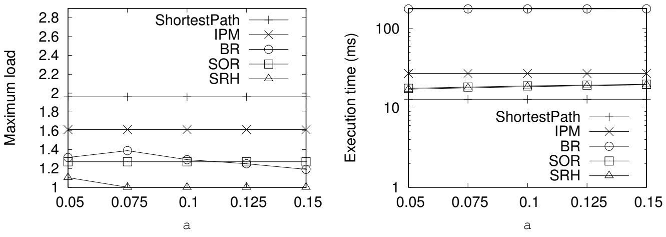
5.2.3 Scalability test. Figure 9 gives the results of the scalability test where we vary the $|Q|$ in the order of 10^5. The average processing time of all the algorithms except *BR* is still low in around 10 ms per query. For the memory cost, they are stable since the dominant



(a) Max. load of varying a (NYC)

(b) Time of varying a (NYC)

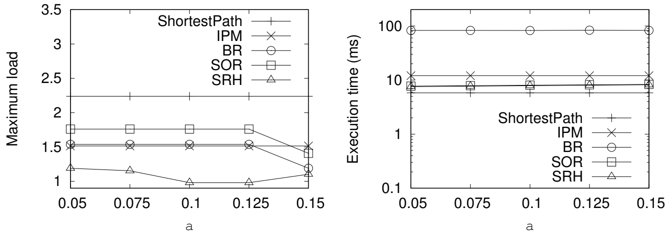
Figure 11: Varying a on NYC



(a) Max. load of varying a (Baidu)

(b) Time of varying a (Baidu)

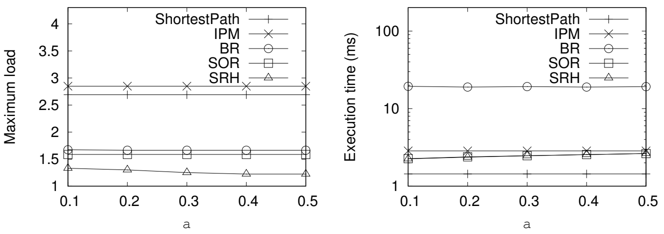
Figure 12: Varying a on Baidu



(a) Max. load of varying a (Didi)

(b) Time of varying a (Didi)

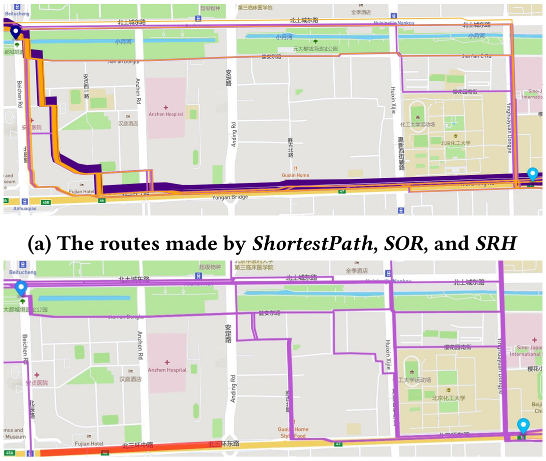
Figure 13: Varying a on Didi



(a) Max. load of varying a (NYC)

(b) Time of varying $|Q|$ (NYC)

Figure 14: Varying a large a



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

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

Figure 15: A case study of the Beijing network

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 $O(mT)$) since we need to check the edge load of each edge-step pair. *SRH* costs double the memory of the other ones because when checking if a pair is in the candidate set C, we simply use boolean values for all pairs, with the same size of $O(mT)$ as the edge loads. In conclusion, all the algorithms are scalable no matter how we increase $|Q|$.

5.2.4 *Effect of small a.* Figure 11, 12, and 13 present the results of varying a small detour factor a on NYC, Baidu, and Didi datasets, respectively. For the maximum load, *ShortestPath* and *IPM* remain unchanged since they do not consider any detour in routing. We could see a generally downward trend for the other three algorithms when a is larger. This is because we allow more detour costs and give the algorithms more path choices. It is possible that there are some small fluctuations or no improvement. We discover that it is hard to further optimize the performance due to the existence of some "bottleneck" edges. They always appear in the resulting paths for some queries no matter how we change a (*e.g.*, several queries have the same origin with only one incident edge.). A small increase of a may change some path choices and make the solution fall into local optimum. *SRH* still achieves the smallest maximum load on all datasets. For the execution time, we could observe similar results to those in varying $|Q|$.

5.2.5 *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 14.

For the maximum load, it could be found that only *SRH* gradually decreases when a is larger. There could be a plateau for *SOR* and *BR* after some point. The reason is similar to the previous one of the bottleneck edges. It also shows 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.

5.3 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 am to 7:30 am (during the morning rush hour) on April 21st, 2017 from Baidu Maps five times. The routes of the *ShortestPath*, *SOR*, and *SRH* are shown in Figure 15a by dark purple, orange, and pink lines, respectively. It can be seen that *ShortestPath* chooses the one with the shortest travel time for 20 vehicles and results in the most crowded road segments. SOR distributes some vehicles to other road segments but still concentrates many on the segments in the left part. *SRH* scatters all the vehicles and guides 9 vehicles differently to use some other road segments to reduce congestion. The recommended routes with limited detour costs are also 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 15b. We could observe that *SRH* (shown in pink lines) could handle such change by avoiding using it and preferring other routes.

5.4 Summary

(i) Our proposed *SRH* which uses history outperforms the state-of-the-art baselines. It could reduce up to 20% the maximum load of *BR*, 40% that of *IPM*, and 50% that of *ShortestPath*.

(ii) SOR and *SRH* are efficient in terms of the average processing time (around 10 ms per query). They are also scalable to large data.

(iii) The proposed *SRH* performs well in the real-world scenario and responds to traffic change immediately.

6 RELATED WORK

In this section, we review related work from two aspects: routing for shortest travel time and congestion minimization.

6.1 Routing for Shortest Travel Time

The basic algorithm for finding the shortest paths on road networks was Dijkstra's algorithm [21]. State-of-the-art solutions constructed precomputed indexes to improve query efficiency. The indexes could be used to either prune the search space [20, 25, 50] or store distances as labels for quick lookups [7, 12, 44]. To handle the real-time updates of edge weights, some directly reduced the time complexity of maintaining indexes [29, 45, 56, 59], while others modeled edge weights by predictable time-dependent functions [33, 37, 53, 54, 60]. However, they all belong to *selfish routing* (*i.e.*, finding the shortest travel time), illustrated below.

A line of work analyzed the inefficiency caused by selfish routing. It was first formulated by the Price of Anarchy (PoA), defined as the ratio of the congestion cost over that of a globally optimal strategy [34, 47]. Bad examples and the first model for analyzing the worst PoA were proposed [48]. Later studies considered the model under different cases: differentiable and convex latency functions [49], those of irregular classes [17], some assumption about the heavy traffic [16], or unifying both the light and heavy traffic [15]. However, they assume that pairs of origins and destinations are given beforehand and try to analyze the inefficiency under different traffic models but not to influence and guide drivers. Our problem is to process each online query so as to guide drives.

6.2 Routing for Congestion Minimization

Alternative routes, in contrast to the shortest ones, were believed to be beneficial for congestion minimization. Early methods included Yen's algorithm [58] and Eppstein's algorithm [22]. Recent techniques were based on the penalty [6, 13], the plateaus [5, 35], and the dissimilarity [14, 39] (see [36] for a thorough study). However, one main issue for them is that there is no common opinion on a "good" route. Their recommended routes may be indifferent to the optimization goal and hence suboptimal in our problem. Our proposed algorithms find the routes which minimize the maximum edge load with theoretical guarantees.

In the transportation area, congestion avoidance strategies were discussed on a small part of the network, such as some junctions (see [27] for a summary). Some proposed re-routing heuristics about some congestion signs [40, 46], and others considered drivers as individual agents and their interactions [19, 52]. They usually modeled many detailed traffic elements, which weaken their scalability (as shown in our experiments). Their experiments were often evaluated on a network with its number of edges less than 10,000. Our problem could process each query in a streaming way and on a large road network. Furthermore, their solutions are heuristic without theoretical guarantees.

In telecommunication networks, online algorithms were proposed to improve the congestion [8, 11, 26, 28, 51]. Specifically, their problems found "Virtual Circuits" routes which, like currents in circuits, occupy all the edges in the routes in an instant and last forever. In [8], an optimal $O(\ln n)$-competitive algorithm was proposed to minimize the maximum load. Extension of the techniques was applied to the circuits with known survival time [9] and reviewed as a primal-dual approach [11]. Other objectives could be minimizing the average latency [28], maximizing the total throughput [26], or minimizing the load under a different input model [51]. In their problems, the route occupies all the edges along it in an instant and lasts forever, but in our problem, each route can only occupy just one edge at any time step and will disappear once the vehicle reaches the destination. Moreover, online packet routing aimed to fulfill the demand of transmitting packets with hard deadlines in a multi-hop network [10, 41, 43]. They are different from our problem because packets could expire and stop at some node, whereas our queries do not have the deadline constraint.

It is also worth noting that some studies focused on the estimation of the future traffic. They used the Bayesian network [23], hidden Markov model [57], autoregressive model [42], crowdsourcing approach [32], or sparse data [30, 55]. All related studies that give confidence bounds could be plugged into the second SRH algorithm since we only use probabilistic bounds to generate a candidate set. Our estimation method is tailored to our algorithm.

7 CONCLUSION

This paper studies a new type of routing queries aiming at minimizing traffic congestion. Particularly, we need to identify the load of each edge (incurred by the routes) in the temporal dimension. We first propose the Spatiotemporal Routing Problem (SRP). To address it, we design the algorithm called Spatiotemporal Oblivious Routing (SOR) with a theoretical guarantee of $O(\ln n)$ in terms of the maximum edge load, where n is the number of nodes. Noticing the recurring property of traffic congestion, we further optimize the SOR by focusing more on some future congested road segments and propose the Spatiotemporal Routing with History (SRH) with a ratio of $O(\ln |C|)$, where C is the set of potential congested road segments derived from history. We also evaluate our algorithms by extensive experiments on three real-world datasets. For future work, one may consider different objectives, such as minimizing the average congestion status, or optimizing different traffic elements (*e.g.*, traffic lights).

REFERENCES

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

[2] 2022. Level of service. https://en.wikipedia.org/wiki/Level_of_service_(transportation).

[3] 2022. Taffic API of Baidu Maps. https://lbsyun.baidu.com/index.php?title=webapi/traffic.

[4] 2022. Trend of using navigation apps. https://www.emarketer.com/content/people-continue-to-rely-on-maps-and-navigational-apps-emarketer-forecasts-show.

[5] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2013. Alternative routes in road networks. *ACM J. Exp. Algorithmics* (2013).

[6] Vedat Akgün, Erhan Erkut, and Rajan Batta. 2000. On finding dissimilar paths. *Eur. J. Oper. Res.* (2000).

[7] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. 2014. Fast Shortest-path Distance Queries on Road Networks by Pruned Highway Labeling. In *ALENEX*, Catherine C. McGeoch and Ulrich Meyer (Eds.).

[8] James Aspnes, Yossi Azar, Amos Fiat, Serge A. Plotkin, and Orli Waarts. 1997. On-line routing of virtual circuits with applications to load balancing and machine scheduling. *J. ACM* (1997).

[9] Baruch Awerbuch, Yossi Azar, and Serge A. Plotkin. 1993. Throughput-Competitive On-Line Routing. In *FOCS*.

[10] Ron Banner and Ariel Orda. 2007. Multipath routing algorithms for congestion minimization. *IEEE/ACM Transactions on Networking* (2007).

[11] Niv Buchbinder and Joseph Naor. 2006. Improved Bounds for Online Routing and Packing Via a Primal-Dual Approach. In *FOCS*.

[12] Zitong Chen, Ada Wai-Chee Fu, Minhao Jiang, Eric Lo, and Pengfei Zhang. 2021. P2H: Efficient Distance Querying on Road Networks by Projected Vertex Separators. In *SIGMOD*.

[13] Dan Cheng, Olga Gkountouna, Andreas Züfle, Dieter Pfoser, and Carola Wenk. 2019. Shortest-Path Diversification through Network Penalization: A Washington DC Area Case Study. In *SIGSPATIAL*.

[14] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, Ulf Leser, and David B. Blumenthal. 2020. Finding k-shortest paths with limited overlap. *VLDB J.* (2020).

[15] Riccardo Colini-Baldeschi, Roberto Cominetti, Panayotis Mertikopoulos, and Marco Scarsini. 2020. When Is Selfish Routing Bad? The Price of Anarchy in Light and Heavy Traffic. *Operation Research* (2020).

[16] Riccardo Colini-Baldeschi, Roberto Cominetti, and Marco Scarsini. 2016. On the Price of Anarchy of Highly Congested Nonatomic Network Games. In *SAGT*.

[17] José R. Correa, Andreas S. Schulz, and Nicolás E. Stier Moses. 2004. Selfish Routing in Capacitated Networks. *Mathematics of Operations Research* (2004).

[18] Stacy Davis and Susan Diegel. 2019. Transportation Energy Data Book: Edition 22. *DIANE Publishing* (2019).

[19] Prajakta Desai, Seng W Loke, Aniruddha Desai, and Jugdutt Singh. 2013. CARAVAN: Congestion avoidance and route allocation using virtual agent negotiation. *IEEE Transactions on Intelligent Transportation Systems* 14, 3 (2013), 1197–1207.

[20] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2014. Customizable Contraction Hierarchies. In *SEA*.

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

[22] David Eppstein. 1998. Finding the k Shortest Paths. *SIAM J. Comput.* (1998).

[23] Xiang Fei, Chung-Cheng Lu, and Ke Liu. 2011. A bayesian dynamic linear model approach for real-time short-term freeway travel time prediction. *Transportation Research Part C: Emerging Technologies* (2011).

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

[25] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. 2012. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transp. Sci.* 46, 3 (2012), 388–404.

[26] Ashish Goel, Monika Rauch Henzinger, and Serge A. Plotkin. 1998. Online Throughput-Competitive Algorithm for Multicast Routing and Admission Control. In *SODA*.

[27] Sara El Hamdani and Nabil Benamar. 2017. A Comprehensive Study of Intelligent Transportation System Architectures for Road Congestion Avoidance. In *UNet*.

[28] Prahladh Harsha, Thomas P. Hayes, Hariharan Narayanan, Harald Räcke, and Jaikumar Radhakrishnan. 2008. Minimizing average latency in oblivious routing. In *SODA*.

[29] Takanori Hayashi, Takuya Akiba, and Ken-ichi Kawarabayashi. 2016. Fully Dynamic Shortest-Path Distance Query Acceleration on Massive Networks. In *CIKM*.

[30] Ryan Herring, Aude Hofleitner, Pieter Abbeel, and Alexandre M. Bayen. 2010. Estimating arterial traffic conditions using sparse probe data. In *ITSC*.

[31] Wassily Hoeffding. 1963. Probability inequalities for sums of bounded random variables. *J. Amer. Statist. Assoc.* (1963).

[32] Huiqi Hu, Guoliang Li, Zhifeng Bao, Yan Cui, and Jianhua Feng. 2016. Crowdsourcing-based real-time urban traffic speed estimation: From trends to speeds. In *ICDE*.

[33] Evangelos Kanoulas, Yang Du, Tian Xia, and Donghui Zhang. 2006. Finding Fastest Paths on A Road Network with Speed Patterns. In *ICDE*.

[34] Elias Koutsoupias and Christos H. Papadimitriou. 1999. Worst-case Equilibria. In *STACS*.

[35] Lingxiao Li, Muhammad Aamir Cheema, Mohammed Eunus Ali, Hua Lu, and David Taniar. 2020. Continuously Monitoring Alternative Shortest Paths on Road Networks. *PVLDB* (2020).

[36] Lingxiao Li, Muhammad Aamir Cheema, Hua Lu, Mohammed Eunus Ali, and Adel N Toosi. 2021. Comparing alternative route planning techniques: A comparative user study on Melbourne, Dhaka and Copenhagen road networks. *IEEE Trans. Knowl. Data Eng.* (2021).

[37] Lei Li, Sibo Wang, and Xiaofang Zhou. 2019. Time-Dependent Hop Labeling on Road Network. In *ICDE*.

[38] Binbing Liao, Jingqing Zhang, Chao Wu, Douglas McIlwraith, Tong Chen, Shengwen Yang, Yike Guo, and Fei Wu. 2018. Deep Sequence Learning with Auxiliary Information for Traffic Prediction. In *KDD*.

[39] Huiping Liu, Cheqing Jin, Bin Yang, and Aoying Zhou. 2018. Finding Top-k Shortest Paths with Diversity. *IEEE Trans. Knowl. Data Eng.* (2018).

[40] Ruilin Liu, Hongzhang Liu, Daehan Kwak, Yong Xiang, Cristian Borcea, Badri Nath, and Liviu Iftode. 2016. Balanced traffic routing: Design, implementation, and evaluation. *Ad Hoc Networks* 37 (2016), 14–28.

[41] Xin Liu, Weichang Wang, and Lei Ying. 2019. Spatial-temporal routing for supporting end-to-end hard deadlines in multi-hop networks. *Performance Evaluation* (2019).

[42] Wanli Min and Laura Wynter. 2011. Real-time road traffic prediction with spatio-temporal correlations. *Transportation Research Part C: Emerging Technologies* (2011).

[43] Michael J. Neely. 2011. Opportunistic scheduling with worst case delay guarantees in single and multi-hop networks. In *INFOCOM*.

[44] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When Hierarchy Meets 2-Hop-Labeling: Efficient Shortest Distance Queries on Road Networks. In *SIGMOD*.

[45] Dian Ouyang, Long Yuan, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Efficient Shortest Path Index Maintenance on Dynamic Road Networks with Theoretical Guarantees. *PVLDB* (2020).

[46] Juan Pan, Iulian Sandu Popa, Karine Zeitouni, and Cristian Borcea. 2013. Proactive vehicular traffic rerouting for lower travel time. *IEEE Transactions on vehicular technology* 62, 8 (2013), 3551–3568.

[47] Christos H. Papadimitriou. 2001. Algorithms, games, and the internet. In *STOC*.

[48] Tim Roughgarden and Éva Tardos. 2002. How bad is selfish routing? *J. ACM* (2002).

[49] Tim Roughgarden and Éva Tardos. 2004. Bounding the inefficiency of equilibria in nonatomic congestion games. *Games and Economic Behavior* (2004).

[50] Peter Sanders and Dominik Schultes. 2012. Engineering highway hierarchies. *ACM J. Exp. Algorithmics* 17, 1 (2012).

[51] Nguyen Kim Thang. 2019. A Competitive Algorithm for Random-Order Stochastic Virtual Circuit Routing. In *ISAAC*.

[52] Shen Wang, Soufiene Djahel, and Jennifer McManis. 2014. A multi-agent based vehicles re-routing system for unexpected traffic congestion avoidance. In *ITSC*.

[53] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. 2015. Efficient Route Planning on Public Transportation Networks: A Labelling Approach. In *SIGMOD*.

[54] Yong Wang, Guoliang Li, and Nan Tang. 2019. Querying Shortest Paths on Time Dependent Road Networks. *PVLDB* (2019).

[55] Yilun Wang, Yu Zheng, and Yexiang Xue. 2014. Travel time estimation of a path using sparse trajectories. In *KDD*.

[56] Victor Junqiu Wei, Raymond Chi-Wing Wong, and Cheng Long. 2020. Architecture-Intact Oracle for Fastest Path and Time Queries on Dynamic Spatial Networks. In *SIGMOD*.

[57] Bin Yang, Chenjuan Guo, and Christian S. Jensen. 2013. Travel Cost Inference from Sparse, Spatio-Temporally Correlated Time Series Using Markov Models. *PVLDB* (2013).

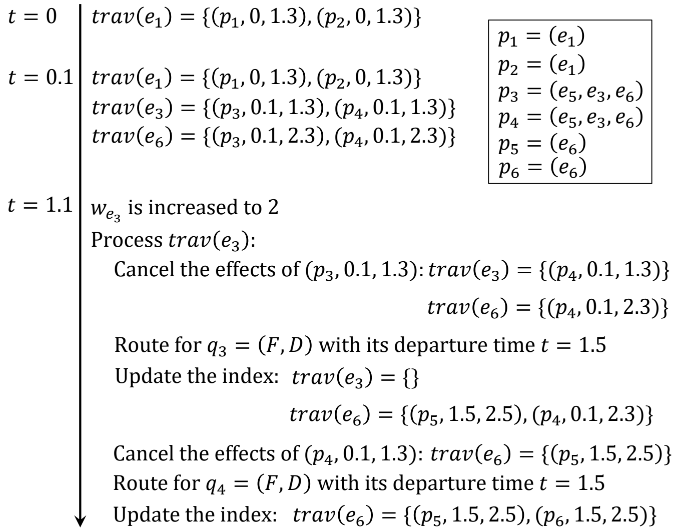
[58] Jin Y Yen. 1971. Finding the k shortest loopless paths in a network. *Management Science* (1971).

[59] Ziqiang Yu, Xiaohui Yu, Nick Koudas, Yang Liu, Yifan Li, Yueting Chen, and Dingyu Yang. 2020. Distributed Processing of k Shortest Path Queries over Dynamic Road Networks. In *SIGMOD*.

[60] Ye Yuan, Xiang Lian, Guoren Wang, Yuliang Ma, and Yishu Wang. 2019. Constrained Shortest Path Query in a Large Time-Dependent Graph. *PVLDB* (2019).

Algorithm 5: 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 edge loads $l(e, \tau)$, and the index $trav(e)$
5     route a new path for this affected query with its new
        origin based on the new edge weight
6     update the variables $x(e, \tau)$, the edge loads $l(e, \tau)$, and
        the index $trav(e)$.
```



$t = 0$ $trav(e_1) = \{(p_1, 0, 1.3), (p_2, 0, 1.3)\}$

$p_1 = (e_1)$
$p_2 = (e_1)$
$p_3 = (e_5, e_3, e_6)$
$p_4 = (e_5, e_3, e_6)$
$p_5 = (e_6)$
$p_6 = (e_6)$

$t = 0.1$ $trav(e_1) = \{(p_1, 0, 1.3), (p_2, 0, 1.3)\}$
$trav(e_3) = \{(p_3, 0.1, 1.3), (p_4, 0.1, 1.3)\}$
$trav(e_6) = \{(p_3, 0.1, 2.3), (p_4, 0.1, 2.3)\}$

$t = 1.1$ w_{e_3} is increased to 2
Process $trav(e_3)$:
Cancel the effects of $(p_3, 0.1, 1.3)$: $trav(e_3) = \{(p_4, 0.1, 1.3)\}$
$trav(e_6) = \{(p_4, 0.1, 2.3)\}$
Route for $q_3 = (F, D)$ with its departure time $t = 1.5$
Update the index: $trav(e_3) = \{\}$
$trav(e_6) = \{(p_5, 1.5, 2.5), (p_4, 0.1, 2.3)\}$
Cancel the effects of $(p_4, 0.1, 1.3)$: $trav(e_6) = \{(p_5, 1.5, 2.5)\}$
Route for $q_4 = (F, D)$ with its departure time $t = 1.5$
Update the index: $trav(e_6) = \{(p_5, 1.5, 2.5), (p_6, 1.5, 2.5)\}$

Figure 16: Edge weight update

A APPENDIX

A.1 Dynamic Edge Weights

Since the edge weight representing the travel time could change in a dynamic road network [45, 56], we will discuss how to handle such edge weight updates. The main idea is to build an index to efficiently find the affected queries and then use the routing algorithm to process them as new queries with new origins. After we handle the update, the variables and the edge loads are correctly maintained as the new edge weight suggests.

When the weight $w_{e'}$ of an edge e' is updated at some time $t' \in \mathbb{R}_{\geq 0}$, two types of queries are not affected. The first one is the future query since our algorithms SOR and SRH all find routes based on the current snapshot of edge weights. Specifically, when we assess a path p by using the sum of variables *w.r.t.* those edge-step pairs that the path p traverses as the metric value, we use the current (updated) edge weights to check the traversed pairs. The second one 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 the edge 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 (line 11 in Algorithm 1 and line 17 in Algorithm 4). We can simply iterate the edges in the path p and compute the time t_{arr} of each edge. Note that if the edge weight is so small that the path p traversing the edge does not show at any step, we can omit the corresponding triples (as e_5 in Example 6) since no loads and variables will be affected.

Example 12. *Back to Example 8, suppose that we use SRH to return paths for queries. At time $t = 0$, 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 time 1.3. The procedure is also shown in Figure 16. At time $t = 0.1$, 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 do not add the triples to $trav(e_5)$ because w_{e_5} is so small that p_3 and p_4 which traverse e_5 do not show at any step.*

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 $trav(e')$ for this edge 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}) which is currently traversing the edge e or will traverse it, 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 routing queries with a new origin, the new departure time, and the old destination, and finally maintain the variables, the loads and the index by the newly returned path. To cancel the effect on the index $trav(e)$, we iterate p's edges based on the original edge weight before the update and remove the corresponding triples in the index. To cancel the effect on the edge load and variables, we iterate the edge-step pairs that p traverses. Based on the newly updated edge weight, we reroute a new path for this query with its current location as the new origin (discussed later). Finally, we maintain the variables, the loads, and the index as previously stated.

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

Algorithm 5 summarizes the whole 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 variables, the loads, and the index in line 6.

Example 13. *At time $t = 1.1$, suppose that w_{e_3} is increased to 2. We first prune the triples with $t_{arr} < 1.1$ in $trav(e_3)$. For the triple $(p_3, 0.1, 1.3)$, we cancel its effect on the index $trav(e)$ by removing $(p_3, 0.1, 1.3)$ from e_3 and $(p_4, 0.1, 2.3)$ from e_6. We then route $q_3 = (F, D)$ where F is the destination node of e_3. Its new departure time is*

Table 3: Effects of δ

δ	0.01	0.05	0.1	0.15	0.2	0.25
Maximum load	1.69	1.61	1.42	1.46	1.46	1.47

calculated as $1.1 + 0.2 \times 2/1 = 1.5$*, where the remaining time on the edge* e_3 *is* $0.1 + 0.2 + 1 - 1.1 = 0.2$*. Since we return path* $p_5 = (e_6)$ *to this new query, we next update the index by inserting the triple* $(p_5, 1.5, 2.5)$ *with its* $t_{arr} = 1.5 + 1 = 2.5$*. The update procedures for loads and variables are omitted. We do the similar procedure for the triple* $(p_4, 0.1, 1.3)$.

The space cost of the entire index is $O(|Q|U)$, where U is the maximum time span. This is because each query could only be stored in different $trav(e)$ at most U times. However, the space cost is much smaller in practice because we remove the obsolete triples constantly so that only the queries in a limited period will be stored. The time complexity of Algorithm 5 is $O(U \max_e trav(e))$ since we examine each triple in $trav(e)$ and for each trip, the time cost of canceling and updating the index, the variables, and loads are all $O(U)$. The time cost is also smaller in practice for the same reason above.

A.2 Experiment Results

A.2.1 Dynamic weights.

A.2.2 Parameters δ and β. Since only the SRH use the parameter δ to generate the candidate set C of edge-step pairs, we show the maximum loads of SRH under different values of δ in Table 3. We use the default setting for the NYC dataset.

It can be seen that $\delta = 0.1$ is the best value with the minimum maximum load. Either very small or large δ are not beneficial. For small δ, the confidence radius is very large so that many edge-step pairs are included in the candidate set C. Too many edge-step pairs would make the algorithm uncertain about the true one that it should avoid using. In the extreme case where all the edge-step pairs are considered, the SRH will degenerate to the SOR. Very large δ indicates a small confidence radius, but δ is also the probability of the event that the random variable falls into the interval, which may be wrong about the edge-step pair with the maximum load.

The maximum load of varying β from 1 to 10 are all 1.42. It is not sensitive to β for two reasons. First, we use the sample covariance to find the confidence radius which could easily be large because of the insufficient samples or the mismatch at some positions of the sequences. Hence, there are few correlated pairs of edge-step pairs for a large covariance and strict β. Second, it is used to make the candidate set C more comprehensive by including some edge-step pairs that are correlated with those already in C (generated by Algorithm 2). If two edge-step pairs are very correlated, they could have been added into the C in Algorithm 2.