Towards Alleviating Traffic Congestion: Optimal Route Planning for Massive-Scale Trips

Ke Li, Lisi Chen and Shuo Shang

University of Electronic Science and Technology of China, China
like_like@std.uestc.edu.cn, lchen012@e.ntu.edu.sg, jedi.shang@gmail.com

Abstract

We investigate the problem of optimal route planning for massive-scale trips: Given a traffic-aware road network and a set of trip queries $Q$, we aim to find a route for each trip such that the global travel time cost for all queries in $Q$ is minimized. Our problem is designed for a range of applications such as traffic-flow management, route planning and congestion prevention in rush hours. The exact algorithm bears exponential time complexity and is computationally prohibitive for application scenarios in dynamic traffic networks. To address the challenge, we propose a greedy algorithm and an $\epsilon$-refining algorithm. Extensive experiments offer insight into the accuracy and efficiency of our proposed algorithms.

1 Introduction

With the continued development of GPS-enabled devices (e.g., vehicle navigation systems and smart phones) and online map-based services (e.g., Google Maps), route search and planning have attracted significant attention in recent years (e.g., [Dumitrescu and Boland, 2003; Sharifzadeh et al., 2008; Levin et al., 2010; Shang et al., 2013b; Li et al., 2013; Zeng et al., 2015; Liang and Wang, 2018; Xu et al., 2017]). Some studies on this topic aim to generate a route based on an optimization goal (e.g., shortest distance, minimum travel time) under user-defined constraints (e.g., source and destination) over traffic-aware road networks [Cao et al., 2012; Shang et al., 2013a; Shang et al., 2015; Liebig et al., 2017; Shang et al., 2016].

However, these studies find an optimal route only for a single trip query. As online map-based services are becoming increasingly popular, it is not uncommon that a great number of users may issue trip queries simultaneously or within a short period of time, especially during rush hours. In such case, recommending the optimal route based on each individual query may induce potential traffic congestion. Let us consider Figure 1 as an example. Let trips 1–4 be four trip queries that are issued simultaneously. Each trip query consists of a source (marked with a circle) and a destination (marked with a diamond). Vertices $v_1$, $v_2$, $v_3$, and $v_4$ are the sources of trips 1–4. Vertices $v_5$, $v_6$, $v_7$, and $v_8$ are the destinations of trips 1–4. Routes $\pi_1$, $\pi_2$, $\pi_3$, and $\pi_4$ are optimal travel routes for trip queries 1–4, respectively. We see that all optimal routes will travel between vertices $v_9$ and $v_{10}$, which may lead to traffic congestion. In turn, the congestion may delay the travel times of all routes that cover the segment between $v_9$ and $v_{10}$.

Consequently, given a large number of trip queries issued simultaneously, we need to regard these queries as a whole and generate a route planning for each query such that the global travel time for all queries is minimized. In particular, we can adjust the optimal routes of trip 2, trip 3, and trip 4 to $\pi_3$, $\pi_5$, and $\pi_7$, respectively. Even if they are sub-optimal results for individual queries, traffic congestion between $v_9$ and $v_{10}$ can be alleviated and the global travel time of all trips 1–4 is improved.

In this light, we study a novel Global Optimal Route (GOR) problem: Given a traffic-aware road network and a set of trip query $Q$, we generate a travel route for each query such that the sum of travel times for all queries is minimized. In our settings, travel time of each edge depends on both the edge’s minimum travel time and flow of vehicles, the query finds a route combination (set of routes) such that global travel time is minimized. The GOR problem has a broad range of application, including traffic management, route planning and congestion alleviation. By processing all trip queries issued within a short period of time in a batch mode, we are able to reduce the average travel time and avoid traffic congestion, especially during commute hours and hol-
days. The problem is challenging due to its high computational cost. The time complexity of our straightforward exact algorithm is exponential to the number of queries. As a result, it is impossible to find the optimal solution in interaction time. To answer the problem efficiently, we first propose a greedy algorithm. Based on the greedy algorithm, an $\epsilon$-refining algorithm and corresponding pruning techniques are developed to further decrease the global travel time effectively. Based on our experimental results, our proposal is capable of achieving high efficiency and high efficacy when processing 10,000 trip queries simultaneously.

**Related work.** A host of studies were developed to answer optimal route queries under different settings [Nikolova, 2006; Li et al., 2007; Ding et al., 2008; Lim et al., 2009; Shang et al., 2013a; Chen et al., 2020a; Chen et al., 2020b; Chen et al., 2019]. These studies only consider the optimization of individual route query. Existing studies of traffic-based global optimization problems [Dafermos and Sparrow, 1969; Dafermos and C., 1972; Lim and Rus, 2012; Dotoli et al., 2014; Babak et al., 2018] can be regarded as the flow assignment problem rather than real-time route planning problem, where the routes are pre-defined and they search for an optimal flow assignment to each route. The queries proposed by these studies are based on a single trip. Hence, their solutions cannot be used to answer our problem. Additionally, time-dependent road network [Peeta and Mahmassani, 1995; Cai et al., 1997; Ding et al., 2008; Wang and Ran, 2012; Huang et al., 2017] is a representative dynamic network, where each edge has different weight at different time, while in our work the weight (travel time) depends on the real-time flow, thus the optimization methods cannot be used in our problem. To the best of our knowledge, none of the existing algorithms can address our problem as they either target a single query or target different settings.

### 2 Preliminaries and Problem Statement

**Traffic-aware road networks.** We define a traffic-aware road network by a connected and directed graph $G(V, E, t_m)$, which consists of a set of vertices $V$ representing road intersections and a set of edges $E$ representing road segments. Each edge $e(v_i, v_j) \in E$ connects two end-points $v_i$ and $v_j$ where $v_i, v_j \in V$. Function $t_m : E \mapsto R$ assigns a real-valued weight $t_m(e)^1$ to edge $e$ that denotes the minimum travel time on road segment $e$ (i.e., the travel time of $e$ when there are no vehicles).

**Traffic flow.** In this paper, traffic flow is defined as the number of vehicles on an edge $e$ at a particular time $t$, denoted by $f_e$, is computed by Equation 1.

$$f_e(t) = f'_e(t) + f''_e(t)$$  \hspace{1cm} (1)

Here, $f'_e(t)$ denotes the number of non-query vehicles on edge $e$ at time $t$ while $f''_e(t)$ is the number of vehicles using our query system. Note that how to compute $f'_e(t)$ is beyond the scope of this paper, and the solutions we proposed are independent of the initial traffic and the original number of vehicles that traverse the segment but do not use the query system.

**Time-flow function.** The time-flow function is used to compute actual travel time of an edge if we take into account the traffic flow. Following existing studies [Lim and Rus, 2012; Babak et al., 2018], we use two popular time-flow functions (cf. Equations 2 and 3) to compute the actual travel time of segment $e$, which is denoted by $t(e)$.

$$t(e) = t_m(e) \times (1 + \alpha \times f'_e)$$  \hspace{1cm} (2)

$$t(e) = t_m(e) \times (1 + \sigma \times (\frac{f'_e}{\varphi}))$$  \hspace{1cm} (3)

Here, $f_e$ denotes the number of vehicles on segment $e$, $\alpha$, $\sigma$, and $\varphi$ are road-specific parameters, which are determined by segment attributes (e.g., speed limit, road width). Note that our proposal is independent of the values of these parameters. How to set appropriate values for these parameters is beyond the scope of our study.

**Route and travel time.** A route $\pi$ is defined by a finite sequence of vertices $(v_1, v_2, \ldots, v_k)$. Denoted by $T(\pi)$ the actual travel time of $\pi$ is the sum of actual travel times of each road segment in $\pi$ (cf. Equation 4). Similarly, the minimum travel time of a route $\pi$, denoted by $T_m(\pi)$, is the sum of minimum travel times of each road segment in $\pi$ (cf. Equation 5).

$$T(\pi) = \sum_{i=1}^{k-1} t(e(v_i, v_{i+1}))$$  \hspace{1cm} (4)

$$T_m(\pi) = \sum_{i=1}^{k} t_m(e(v_i, v_{i+1}))$$  \hspace{1cm} (5)

**Problem definition.** Given a traffic-aware road network $G(V, E, t_m)$ and a set of trip queries $Q = \{q_1, q_2, \ldots, q_n\}$ where each query $q_i = \{s_i, d_i, \tau_i\}$ consists of a source location $s_i$, a destination location $d_i$, and a departure time $\tau_i$, the Global Optimal Route (GOR) problem finds an optimal route $\pi_i$ from $s_i$ to $d_i$ for each trip $q_i \in Q$ such that the global travel time for the route combination $\Pi$ (i.e., $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$) is minimized. Note that the global travel time of $\Pi$, denoted by $GT(\Pi)$, is computed by Equation 6.

$$GT(\Pi) = \sum_{i=1}^{n} T(\pi_i)$$  \hspace{1cm} (6)

### 3 Solutions for GOR Problem

#### 3.1 Exact Algorithm

Given a GOR query $Q$, a straightforward method is to evaluate all possible route combinations and select the combination that has the minimum global travel time. Specifically, for each trip query $q_i \in Q$ we run Depth-First Search (DFS) on the road network to find all possible routes from $s_i$ to $d_i$. Here, we employ a pruning technique [Shang et al., 2015] that enables early termination of DFS. Next, for each route combination we compute its exact global travel time. We return the route combination with the minimum global travel time as the result.
Complexity analysis. The exact algorithm evaluates every route combination. Assume that the number of possible routes of a trip is \(p\) and the number of vertices of each route is \(|\pi|\). The total number of route combinations is \(p^{|Q|}\) and we need to visit \(|\pi||Q|\) vertices to compute the global travel time of each route combination. For each vertex, we have to traverse each \(q_i \in Q\) to count current flow on the edge. Hence, the time complexity of exact algorithm is \(O(p^{|Q|}|\pi||Q|^2)\).

### 3.2 Greedy Algorithm

From Section 3.1 we see that the exact algorithm is computationally prohibitive for answering the GOR problem when \(|Q|\) is large. Hence, a greedy algorithm is proposed to answer the GOR problem efficiently. The high-level idea is to generate a global optimal route for each trip query \(q_i \in Q\) concurrently through network expansion.

**Route label.** To enable concurrent network expansion of routes, we maintain a route label \(l(q_i) = \langle v_a, \tau_a, \pi_a \rangle\) for each trip query \(q_i\) that records its evaluation progress. The label \(l(q_i)\) consists of three elements: \(v_a\), currently arrived vertex; \(\tau_a\), the time arriving at \(v_a\); \(\pi_a\), the up-to-date route from \(q_i\)'s source to \(v_a\). Initially, the currently arrived vertex for each trip query \(q_i\) is the source location and arrival time is the departure time of \(q_i\) (i.e., \(\tau_a\)).

**Network expansion strategy.** Given a route label \(l(q_i) = \langle v_a, \tau_a, \pi_a \rangle\), we select an optimal subsequent vertex based on network expansion. In particular, we select an adjacent vertex \(v_j\) of \(v_a\) that has the minimum estimated global travel time. Here, the estimated global travel time for route label \(l(q_i)\), denoted by \(\text{EGT}(l(q_i))\), is computed by Equation 7.

\[
\text{EGT}(l(q_i)) = T(\tau_a) + t(e(v_a, v_j)) + t_m(e(v_j, q_i, d)) \quad (7)
\]

**Algorithm details.** Algorithm 1 presents the pseudo code of our greedy algorithm. Initially, we initialize our route combination set \(\Pi\) and priority queue \(H\) that sorts trip queries in ascending order based on \(l(q_i, \tau_a)\) (line 1). Then we insert all initial route labels of trip queries from \(Q\) to \(H\) (lines 2–5). During the search and update process, in each iteration we select a route label \(l(q)\) from \(H\) and expand its route to next vertex until \(H\) is empty, namely all trip queries are processed to destinations (lines 6–25). To be specific, each time we get the \(l(q)\) with minimum \(l(q), \tau_a\) on the top of \(H\) (line 7), for currently arrived vertex \(l(q), v_a\), we select one of its adjacent vertices \(v\) with minimum \(\text{EGT}(l'(q))\) for expansion (lines 8–18). For each adjacent vertex we check whether it conflicts two thresholds: it cannot appear twice in the trip’s planned route \(l(q), \pi_a\) and its minimum travel time to destination \(d\) cannot be larger than that of \(l(q), v_a\) (line 10). If it satisfies both thresholds, we compute the traffic flow for the edge \(e(l(q), v_a, v)\) and the expected travel time \(\text{EGT}(l'(q))\) (lines 11–12). In the next, if \(\text{EGT}(l'(q))\) is the currently minimum, we update \(l'(q), \tau_a\) and \(l''(q)\) is used to store the \(l'(q)\) with minimum \(\text{EGT}(l'(q))\) (lines 13–16). Once the greedy selection is completed, \(l(q)\) has to be updated (line 19). At the end of each iteration, if the trip query is processed to its destination \(d\), we add \(l(q), \pi_a\) to \(\Pi\), else we insert the up-to-date \(l(q)\) into \(H\) again waiting for next expansion (lines 20–24).

**Algorithm 1 Greedy Algorithm**

Input: Traffic-aware road network \(G(V, E, t_m)\);
A set of trip queries \(Q = \{q_1, q_2, \ldots, q_n\}\);
Output: A route combination \(\Pi = \{\pi_1, \pi_2, \ldots, \pi|Q|\}\);

1: \(H \leftarrow \emptyset; \Pi \leftarrow \emptyset;\)
2: for each trip query \(q_i\) in \(Q\) do
3: \(\text{Initialize } l(q_i);\)
4: \(H\.push(l(q_i));\)
5: end for
6: while \(H \neq \emptyset\) do
7: \(l(q) \leftarrow H\.pop();\)
8: for each adjacent vertex \(v\) of \(l(q), v_a\) do
9: \(l'(q) \leftarrow \langle v, l(q), \tau_a, l(q), \pi_a, \text{add}(v) \rangle;\)
10: if \(v \in l(q), \pi_a\) and \(t_m(v, d) < t_m(l(q), v_a, d)\) then
11: \(\text{Compute the traffic flow on edge } e(l(q), v_a, v);\)
12: \(\text{Compute } \text{EGT}(l'(q));\)
13: if \(\text{EGT}(l'(q))\) is the minimum then
14: \(l'(q), \tau_a \leftarrow l(q), \tau_a + t(e(l(q), v_a, v));\)
15: \(l''(q) \leftarrow l'(q);\)
16: \(\text{end if}\)
17: \(\text{end if}\)
18: end for
19: \(l(q) \leftarrow l''(q);\)
20: if \(l(q), v_a = d\) then
21: \(\Pi\.add(l(q), \pi_a);\)
22: else
23: \(H\.push(l(q));\)
24: \(\text{end if}\)
25: \(\text{end while}\)
26: return \(\Pi\)

### 3.3 \(\varepsilon\)-Refining Algorithm

To improve the result quality, we develop an \(\varepsilon\)-refining algorithm that refines the route combination generated by our greedy algorithm. The high-level idea works as follows: After generating the result route combination \(\Pi\), we perform a particular refining operation, including add, drop, and swap, on each route if the operation can decrease the global travel time of \(\Pi\) at least by a factor of \(1 + \varepsilon\). Specifically, for each route \(\pi_i \in \Pi\) we select a refining operation that produces the maximum decrease of global travel time and check if it is improved by a factor of \(1 + \varepsilon\) (i.e., \(\text{GT}(\Pi')\) is no less than \((1+\varepsilon)\) where \(\Pi'\) denotes the route combination after performing the selected refining operation).

**Refining Operations**

We detail three refining operations: add, drop, and swap. The \(\varepsilon\)-refining algorithm reuses the results of greedy algorithm and refines each route \(\pi \in \Pi\) from the source vertex. We evaluate each route sequence of \(\pi\) (i.e., \(\forall (v_i, v_{i+1}) \in \pi\) where \(i \in [1, |\pi| \geq 1]\)) and perform the following refining operations if possible.

- **drop \((v_i)\):** drop a vertex \(v_i \in \pi\) if \(|\pi| \geq 3\) and an edge exists between \(v_{i-1}\) and \(v_{i+1}\).
- **add \((v)\):** add a vertex \(v \in (V \setminus \pi)\) to \(\pi\) if \(v\) is an intermediate vertex between \(v_i\) and \(v_{i+1}\).
Figure 2: Examples of drop, add, and swap operations

- swap $(v_i, v_j)$: swap a vertex $v_i \in \pi$ with another vertex $v_j \in (V \setminus \pi)$ if more than one alternative intermediate vertices exist between $v_{i-1}$ and $v_{i+1}$. Figure 2 exemplifies drop, add, and swap operations. The blue edges are the primitive route $\pi = (v_1, v_2, v_3)$ generated by the greedy algorithm. The red edges in Figures 2(a), 2(b), and 2(c), are the refined segments after performing drop, add, and swap operations, respectively.

Consider the drop operation in Figure 2(a), if an edge between $v_4$ and $v_5$ exists, the decrease of global travel time is $\Delta GT_{drop} = t(e(v_4, v_1)) + t(e(v_1, v_2)) + t(e(v_2, v_3)) + t(e(v_3, v_4))' - t(e(v_4, v_5))'$. For swap operation in Figure 2(c), where $v_5$ can take place of $v_1$ as an intermediate vertex linking $v_4$ and $v_5$, we adjust the current travel time $\Delta GT_{swap} = t(e(v_4, v_1)) + t(e(v_1, v_2))' + t(e(v_4, v_5))' - t(e(v_5, v_2))'$. However, in order to update the travel time of each trip, we need to compute the current flows of all altered edges each time we perform a refining operation, which is very time-consuming. To improve the efficiency of refining operations, we develop pruning strategy for each type of operation by computing the corresponding upper bound of decreased travel time. In particular, the upper bounds of decreased travel time for drop, add, and swap, denoted by $UB_{drop}$, $UB_{add}$, and $UB_{swap}$, respectively, are computed as follows.

$$UB_{drop} = t(e(v_4, v_1)) + t(e(v_1, v_2)) + t(e(v_2, v_3))' + t(e(v_3, v_4))' \geq \Delta GT_{drop}$$ (8)
$$UB_{add} = c_{e(v_2,v_3)} + c_{e(v_1,v_2)}' \geq \Delta GT_{add}$$ (9)
$$UB_{swap} = t(e(v_4, v_1)) + t(e(v_1, v_2))' + t(e(v_4, v_5))' \geq \Delta GT_{swap}$$ (10)

Thus, for each operation we first compute $UB$ (i.e., $UB_{drop}$, $UB_{add}$, and $UB_{swap}$) to determine whether the operation is valid. If $UB$ cannot decrease the global travel time by a factor of $1+\epsilon$ (i.e., $GT/(GT-UB) \leq 1+\epsilon$), the operation is invalid, else we further compute the exact decreased global travel time $\Delta GT$ (i.e., $\Delta GT_{drop}$, $\Delta GT_{add}$, and $\Delta GT_{swap}$) to compare with other operations.

Algorithm details. Algorithm 2 presents the pseudo code of our $\epsilon$-refining algorithm. The query input includes a traffic-aware road network $G(V,E,t_m)$ and the results of greedy algorithm, a refining factor $\epsilon$ is selected as parameter, while the output is the refined route combination. Initially, we maintain a dynamic priority heap $H$ for selecting the route label $l(q)$ with minimum $l(q), \tau_a$ (line 1). Then, we insert all initial route labels of trip queries from $Q$ to $H$ based on $l(q), \tau_a$ (lines 2–5). During the refining process, we apply a valid operation with maximum decrease of global travel time to update global travel time and corresponding route labels. The process terminates when no new operation is valid to produce a better result, namely all routes’ adjusting progresses are finished (lines 6–18). To be specific, in each iteration, we select the route label $l(q)$ on the top of $H$ (line 7). Consider the current route sequence of $l(q), v_a$, we compute the corresponding upper bound of decreased travel time first to determine whether the operation is valid. If an operation is valid, we further compute the exact decrease (line 8). After selecting the operation with the maximum global travel time decreased (line 9), we apply the operation with maximum decrease to update global travel time $GT$, the route label $l(q)$ and corresponding route in $\Pi$ respectively if the global travel time is improved at least by a factor of $1+\alpha$ (lines 10–14). At the end of each adjustment, if $l(q)$ is adjusted to its destination, it doesn’t need to be refined any more, else we insert up-to-date $l(q)$ into $H$ again for next adjustment. Finally, we return the refined $\Pi$ as the result, denoted by $\Pi'$.
Complexity analysis. Assume \( GT_0 \) is the global travel time of the obtained results of greedy algorithm and \( GT' \) is the global optimal travel time. By utilizing the small constant \( \epsilon \), the maximum number of operations in \( \epsilon \)-refining algorithm \( m \) depends on the ratio of \( GT_0 \) to \( GT' \). The value of \( m \) is computed as follows.

\[
GT_0 \cdot (1 + \epsilon)^{-m} \leq GT' \\
\Rightarrow m = \lceil \log_{1+\epsilon} \frac{GT_0}{GT'} \rceil \tag{11}
\]

To find a valid operation, we check all cases of drop, add, and swap operations in each adjustment. The time complexity is \( O((n_{11} + n_{12} + n_{13})|Q|) + O((n_{21} + n_{22} + n_{23})|Q|) + O((n_{31} + n_{32} + n_{33} + n_{34})|Q|) = O(|Q|^2) \). Here, \( n_{11}, n_{12}, \ldots, n_{34} \) are the number of other influenced trips on altered edges and they are upper to \( |Q| \). Assume the maximum number of vertices of all routes is \( k \) and the number of influenced trips by this adjustment is \( n \), then updating routes and progress records requires time complexity \( O(kn) \). Here, \( m \) is a constant. Thus, the time complexity of the \( \epsilon \)-refining algorithm is \( O(mkn|Q|^2) = O(|Q|^3) \) for \( k \) is much smaller than \( |Q| \) and \( n \) is upper to \( |Q| \).

4 Experimental Study

4.1 Experimental Settings

Compared algorithms. We study the performance of the following proposed algorithms: the greedy algorithm (Section 3.2) and the \( \epsilon \)-refining algorithm (Section 3.3). And to evaluate the result quality of the two algorithms, we implement an individual-based search algorithm. Specifically, given a set of trip queries \( Q \), for each query \( q_j \in Q \) we derive a route \( \pi \) with the minimum travel time (i.e., \( T_{\pi}(\pi) \)). Note that the exact algorithm requires at least 1 day with default setting, thus we do not report its performance. In remaining parts of this paper, the greedy algorithm is denoted by "Gre-Alg.", the \( \epsilon \)-refining algorithm is denoted by "Ref-Alg.", and the individual-based search algorithm is denoted by "Ind-Alg.".

Data and queries. We use two road networks in our experimental study, namely San Joaquin County Road Network (TG) and the New York Road Network (NYN), which contain 18,263 vertices and 23,874 edges, and 95,581 vertices and 260,855 edges, respectively. The road networks are maintained by memory-based adjacency lists. We pre-compute the all-pair minimum \( T_m(\Pi) \) using Dijkstra’s algorithm [Dijkstra, 1959] and store the results on disk. All algorithms are implemented in Java and tested on a Windows 10 platform with Intel(R) Core(TM) i5-9300H Processor (2.40 GHz) and 16GB memory. The sources and destinations are selected randomly. The departure time of each trip is randomly generated within a specific time range. The main performance metrics are CPU time and the global travel time (GT). Unless stated otherwise, the experimental results are averaged over 20 independent trials with different query inputs. We apply Equation 2 (cf. Section 2) as time-flow function by default. The default parameter settings are listed in Table 1.

### Table 1: Parameter Settings

<table>
<thead>
<tr>
<th></th>
<th>TG</th>
<th>NYN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of trip queries ( Q )</td>
<td>2,000-10,000 /default 10,000</td>
<td>4,000-20,000 /default 10,000</td>
</tr>
<tr>
<td>Refining factor ( \epsilon )</td>
<td>0.02-0.10 /default 0.02</td>
<td>0.02-0.10 /default 0.02</td>
</tr>
<tr>
<td>( \alpha ) (Equation 2)</td>
<td>0.02-0.10 /default 0.02</td>
<td>0.02-0.10 /default 0.02</td>
</tr>
<tr>
<td>( \sigma ) (Equation 3)</td>
<td>0.15</td>
<td>0.15</td>
</tr>
<tr>
<td>( \varphi ) (Equation 3)</td>
<td>20-100 /default 20</td>
<td>20-100 /default 20</td>
</tr>
<tr>
<td>( \beta ) (Equation 3)</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

4.2 Experimental Results

Effect of query trip count. First, we investigate the effect of the query trip count \( |Q| \) on the performance of the algorithms with the default setting. Intuitively, a larger \( |Q| \) causes more vertices to be processed (being selected or being adjusted). Thus, a larger \( |Q| \) leads to more computation effort and the CPU time cost is higher for all algorithms. Additionally, a larger \( |Q| \) leads to the increment of traffic flows on edges, and thus increases the travel time. In this case, Gre-Alg and Ref-Alg may reduce the global travel time substantially. Figure 3 shows the CPU time and the global travel time of the results of three algorithms in TG and NYN respectively. As expected, when \( |Q| \) becomes larger, the CPU time increases for all algorithms. Compare with Ind-Alg, the Gre-Alg performs slightly slower. We also find that Ref-Alg requires more CPU time than the other two algorithms due to its update operations in each iteration of refinement. Nevertheless, we can see all algorithms are reasonably efficient. It is clear that the route combination generated by Gre-Alg consistently exhibits less global travel time compared with Ind-Alg. Compared with Ind-Alg, Ref-Alg greatly decreases the global travel time. It is worth noting that there are little traffic flow on each edge when \( |Q| \) is small, in this case the improvements of Gre-Alg and Ref-Alg are negligible (i.e., \( |Q|=2,000 \)).
is because that the performance when traffic flow is large as $|Q|$ increased, the improvement of Gre-Alg and Ref-Alg is significant (i.e., Gre-Alg can reduce over 30% global travel time and Ref-Alg can reduce over 60% global travel time compared with Ind-Alg when $|Q|$ is 20,000 in NYN). These results demonstrate that Gre-Alg and Ref-Alg are effective to alleviate traffic congestion, especially when $|Q|$ is large.

**Effect of $\epsilon$.** Figure 4 shows the efficiency and efficacy performance as we vary $\epsilon$. The total number of operations is inversely proportional to the value of $\epsilon$ (cf. Section 3.3). Intuitively, a larger $\epsilon$ means fewer valid operations, thus less CPU time is required. However, a larger $\epsilon$ also results in more global travel time. The minimum global travel time, denoted by $GT^*$, is the global time of individual optimum when we assume that $f_e = 0$. Though the global optimal travel time $GT'$ is unobtainable, $GT^*$ can take place of $GT'$ to compute an approximation ratio upper bound for $GT^*$ is close to $GT'$. The upper bound of the approximation ratio, denoted by "Approx-Ratio*$", is the ratio of $GT$ to $GT^*$. Thus, the value of "Approx-Ratio*$" is larger than that of actual approximation ratio for $GT^*$ is smaller than $GT'$.

**Effect of pruning techniques.** We study the effect of pruning techniques in Ref-Alg. The Ref-Alg without pruning techniques is denoted by "Ref-Alg*." In Figure 5, With the help of pruning techniques, the performance of Ref-Alg is improved by about an order of magnitude when $|Q|$ is large (i.e. $|Q|$ $\geq$10,000). And as expected, Ref-Alg* only reduce little global travel time compared with Ref-Alg. These results demonstrate the accuracy of our proposed pruning rules.

**Effect of time-flow functions.** To study the scalability of our proposed algorithms, we apply two different time-flow functions in Section 2 (Equations 2 and 3) and vary the corresponding parameters. Intuitively, a larger value of $\alpha$ and a smaller value of $\varphi$ will cause more congestion for increased actual travel time, and the CPU time will increase accordingly. Due to space limit, we only show the results in TG. Figure 6 proves the CPU time and global travel time of the three algorithms when we apply Equation 2 and Equation 3 as our time-flow functions, respectively. It is clear that CPU time increases as we increase $\alpha$ or decrease $\varphi$ as shown in Figure 6(a) and Figure 6(c), respectively. In Figure 6(b), when we vary $\alpha$ from 0.02 to 0.10, an increasing trend regarding global travel time is observed for all methods. However, the increasing trend of Ref-Alg is less significant than the other two methods, demonstrating the stronger capability of Ref-Alg in alleviating traffic congestion and reducing the global travel time. In Figure 6(d), when we vary $\varphi$ from 20 to 100 in Equation 3, the global travel time decreases. The results demonstrate that our proposed algorithms are capable for handling the query for most of settings, and different time-flow functions allow our proposed algorithms to apply to diverse traffic models.

**Worst-Case measurement.** The worst-case measurement is the worst experimental result with the default setting among 20 independent experiments. Figure 7 reports worst-case measurements. We observe that Gre-Alg and Ref-Alg are both able to compute the query in interaction time. Generally, the CPU time is at most 1,200 ms in TG and 2,400 ms in NYN, respectively. Additionally, Ref-Alg can finally reduce at least 42% global travel time and 45% global travel time when $|Q|$ = 10,000 in TG and $|Q|$ = 20,000 in NYN, which demonstrate the two proposed algorithms are both able to effectively alleviate traffic congestion.

5 Conclusion

We proposed and investigated the problem of global optimal route planning for massive-scale trips (GOR problem). To address GOR problem, two algorithms were developed in this paper, and some pruning strategies were established to further enhance the efficiency. Extensive experiments confirmed that all the proposed algorithms are capable of achieving both high efficiency and high effectiveness.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 61932004).
References


