面向时变路网拼车的快速插入算子

Zengyang Gong 香港科技大学 中国香港特别行政区
zgongae@connect.ust.hk

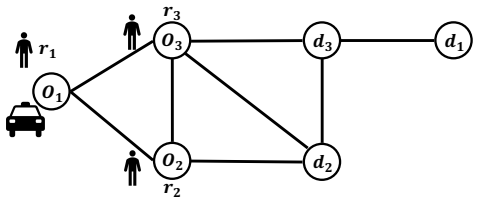
Yuxiang Zeng
香港科技大学
和
中国香港特别行政区
yzengal@cse.ust.hk

Lei Chen
香港科技大学
香港科技大学
中国香港特别行政区
leichen@cse.ust.hk

arXiv:2303.03614v1 [cs.DB] 7 Mar 2023

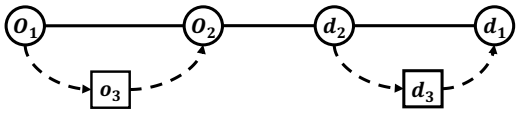
摘要

近年来，拼车因其经济和社会效益，已成为一种颇具前景的出行方式。作为一项重要的算子，“插入算子”已在静态道路网络中得到广泛研究。当出现新请求时，插入算子用于寻找将该请求的起点和目的地插入工作人员当前路线的最优位置，从而使该工作人员的行程时间最短。先前研究探讨了如何在静态道路网络中高效执行插入操作；然而在现实中，路径规划应考虑动态交通场景（即时变道路网络）。遗憾的是，在这种情况下，现有插入算子的解决方案成了拼车路径规划算法的瓶颈，因为工作人员服务新请求的可行性和行程时间取决于该请求的取货和送达时间，而不能直接在静态道路网络上计算。本文研究时变道路网络上的插入算子。具体而言，我们首先提出一种基线插入方法，从头计算沿新路线的到达时间，其耗时为 $O(n^3)$，其中 n 是分配给该工作人员的请求总数。为降低较高的时间复杂度，我们计算沿路线的复合行程时间函数，以加快计算路线中顶点对之间行程时间的速度，因此，插入操作的时间复杂度可降至 $O(n^2)$。最后，我们进一步改进该方法，提出一种线性时间插入算法，并证明在依次枚举每个可能的新请求目的地插入位置时，只需耗时$O(1)$即可找到当前路线中插入起点的最佳位置。在两个真实世界的大规模数据集上的评估表明，我们的方法最多可将现有插入算法的速度提升 25.79 倍。



请求	r_1， #2, #3
请求的起点	o_1， $2, $3
请求目的地	d_1， d_2， d_3
r_1、r_2、r_3的发布时间	上午8:00、上午8:00、上午8:00

(a) 时变道路网络



Edge	权重{（时间，权重）}
o_1o_3	{(&:(()*, ,(, &:)((, /(})
o_3o_2	{(&:(()*, 1 , &):,(-., ,(})
o_2d_2	{(&:(()*, ,1 , &)((., /(, 3:((-., 2( )
d_2d_3	{(&:(()*, 1 , &):4((-., ,(})
d_3d_1	{(&:(()*, ,(, &:)1((., ,(, 3:((-., /))

(b) 插入算子

图 1：时变道路网络上的插入算子。

1 引言

拼车是一项由工作人员提供拼车出行的服务，面向多个请求具有相似的出行安排。在现实世界中，许多广泛存在的应用，如拼车、食品配送和最后一公里物流，都基于拼车服务 [21, 25]。该服务中有一组工作人员和动态到达的请求；当出现新请求时，一项基本功能是找到行程时间最短的工作人员来运送该请求。由于拼车具有巨大的社会和经济效益，例如缓解交通拥堵、提高道路利用率，因此拼车已得到广泛研究 [14], [17], [8]。

插入算子已被视为解决拼车路径规划问题的基础操作，并且在静态道路网络场景中得到了充分研究 [24], [28], [29]。给定一名工作人员及其用于服务已分配请求的可行路线，当出现新请求时，插入算子会尝试在路线中找到适当的位置，以插入新请求的起点和目的地，同时保持路线的可行性，并使新路线的总行程时间最小化。在相关文献中，该算子被广泛用于拼车服务中的路线规划 [17], [14], [24], [23], [28]。

然而，在现实应用中，道路网络中一条边的行程时间会随时间变化。因此，近年来，道路网络被形式化为时变图，其中每条边都关联一个时变权重函数 [26]。现有的插入算子在时变道路网络上运行效率低下，成为拼车服务的瓶颈。因此，为了实时服务请求，本文研究了一种适用于时变道路网络的快速插入算子。

我们通过以下示例说明，在时变道路网络中使用插入算子所面临的困难与挑战。**示例 1.** 图*1a*展示了时变道路网络中的拼车服务示例。共有三个请求 r_1、r_2 和 r_3。上午 *8:00*，又有三个请求 r_1、r_2 和 r_3 发出，并分配给

位于 o_1 的工作人员。我们假设 $\langle o_1, o_2, d_2, d_1\rangle$ 是服务 r_1 和 r_2 的最短时间路线，在图*1b*中以实线表示。对于道路网络中起讫点对为 o_3 和 d_3 的请求 r_3，为服务该请求，插入算子会尝试将新请求的起点和目的地分别插入到 o_2 和 d_1 之前，如图*1b*中的虚线所示。由此可得到新的候选路线 $\langle o_1, o_3, o_2, d_2, d_3, d_1\rangle$。特别地，时变道路网络中路线的每条边都关联着一组时变权重。例如，道路网络中边 $\langle o_1, o_3\rangle$ 关联着一个集合 *{(8:00am, 10),(8:10am, 20)}*，例如，对于 $\langle o_1, o_3\rangle$，*(8:00am, 10)* 表示在上午 *8:00* 从 o_1 到 o_3 需要 *10* 分钟，而 *(8:10am, 20)* 表示在上午 *8:10* 需要 *20* 分钟。动态出行成本使得高效应用插入算子更具挑战性，具体解释如下。

挑战。对于图1b所示、用于服务新请求的候选路线 o_1，o_3，o_2，d_2，d_3，d_1，在研究充分的静态道路网络中，可直接将静态道路网络上4条边的固定行程时间相加，以更新这条路线的行程时间例如，插入 o_3 时经过 o_1，o_3 和 o_3，o_2，插入 d_3 时经过 d_2，d_3 和 d_3，d_1。然而，在进一步计算这条路线的行程时间时，可以忽略路线中每个顶点的到达时间。但在这个时变道路网络示例中，顶点的最新到达时间取决于前一顶点的到达时间。假设从 o_1 出发，时间为 t_1，且 $t_1 = 8:\ 00am$ 我们可以通过递归方式从头计算这条路线的行程时间

$$
\begin{aligned}
arr(o_1) &= t_1 \\
arr(o_3) &= arr(o_1) + disquery(o_1, o_3, arr(o_1)) \\
&\cdots \\
arr(d_3) &= arr(d_2) + disquery(d_2, d_3, arr(d_2)) \\
arr(d_1) &= arr(d_3) + disquery(d_3, d_1, arr(d_3))
\end{aligned}
$$

$arr(o_1)$表示该路线中到达顶点 o_1 的时间，而$disquery(o_1, o_3, arr(o_1))$表示在时变道路网络上的最短旅行时间查询，它返回从 o_1 到 o_3 的最短旅行时间，即从 o_1 在时间 $arr(o_1)$出发时的最短行程时间。该递归方法会产生大量最短旅行时间查询。在大规模时变道路网络上的最短旅行时间查询比在静态道路网络 [26], 上的查询更加耗时，因此频繁调用最短旅行时间查询会使插入算子成为时变道路网络上拼车服务的瓶颈。

本文研究时变道路网络上的拼车插入算子。我们解决方案的主要思想如下。首先，我们研究基线方法 [24]，其时间复杂度为 $O(n^3)$，该方法会在时变道路网络上调用大量最短旅行时间查询，使插入操作成为效率瓶颈。然后，我们通过复合边权函数、检查约束并在 $O(1)$时间内计算目标函数，将时间复杂度降至 $O(n^2)$。最后，我们通过枚举用于插入目的地的路线潜在位置，并在 $O(1)$时间内找到用于插入新请求起点的最优位置，将方法的时间复杂度进一步降至 $O(n)$。

主要贡献总结如下：

表1：主要记号汇总。

记号	描述
$G(V, E, F)$	时变道路网络
$disquery(u, v, t)$	在 $G(V, E, F)$ 上的最短旅行时间查询
$ComTravel(u, v, t)$	路线 u 到 v 上的复合旅行时间函数
v_w, c_w	工作人员的位置和容量
o_r, d_r	请求的起点和终点
$arr(o_r)$ $arr(d_r)$	请求的接载时间和送达时间 r
$arr[k]$	工作人员路线中 v_k 处的到达时间
$num[k]$	工作人员路线中 v_k 处的已接载请求数
$latest[k]$	为保持路线可行性，在 v_k 处的最晚到达时间
$pckl[k]$	i 在 v_k 之前插入 d_r 时的值

- 我们首次研究了适用于时变道路网络拼车的高效插入算子。
- 我们设计了一种新颖的方法，通过组合道路路段的权重函数，将时间复杂度从基线方法中的三次降低到二次。
- 随后，我们提出了一种时间复杂度为 $O(n)$ 的最优解，并将内存开销空间从二次时间方法中的 $O(n^2)$ 降低到 $O(n)$。
- 在真实数据集上的大量实验表明，我们的解决方案可使时变道路网络上的插入算子提速 2.16 至 25.79 倍。

本文其余部分安排如下。我们首先在第2节阐述问题定义。随后，在第3节详细介绍我们的算法。我们在第4节评估算法，并在第5节回顾相关工作。最后，第6节总结全文。

2 问题定义

本节介绍时变道路网络上插入算子的记号和形式化定义。本文的主要记号汇总于表1。

2.1 时变道路网络

定义 1（时变道路网络）。使用有向图 *G(V,E,F)* 对动态路网进行建模，其中 V 是顶点集合，每个顶点 $v \in V$ 表示一个地理位置（例如，道路交叉口）；$E \subset V \times V$ 是道路边集合。对于每条边 $(u, v) \in E$，它表示一条从 u 到 v 的有向边，并关联一个权重函数 $f_{u,v}(t) \in F$；t 是一个变量，表示出发时间。$f_{u,v}(t)$ 的值表示在时间 t 从 u 出发到达 v 所需的行程时间，该值始终非负。

按照现有研究的惯例 [26] [6] [27],，我们使用分段线性函数对时变道路网络建模。为降低空间开销，将一组插值点 $P = \{(t_1, w_1), (t_2, w_2) \ldots, (t_k, w_k)\}$ 保存下来，以拟合 $f_{u,v}(t)$。对于每个点 (t_i, w_i)，它表示从 u出发的时间为 t_i时，到达 v需要 w_i个单位时间，其中任意 $x \le y, t_x \le t_y$均适用。连接两个相邻点 (t_i, w_i)、(t_{i+1}, w_{i+1}) 的直线拟合时间域 $[t_i, t_{i+1}]$上的线性函数。例如，如果工作人员在 $t \in [t_1, t_2]$从 u出发，其行程时间将为 $f_{u,v}(t) = w_1 + (t - t_1)\frac{w_2 - w_1}{t_2 - t_1}$。于是，与 (u, v) 相关联的时间函数可形式化为——

$$f_{u,v}(t)=\begin{cases} w_1, & t=t_1 \\ w_1+(t-t_1)\frac{w_2-w_1}{t_2-t_1}, & t_1<t\le t_2 \\ \cdots & \\ w_{k-1}+(t-t_{k-1})\frac{w_k-w_{k-1}}{t_k-t_{k-1}}, & t_{k-1}<t\le t_k \\ w_k, & t=t_k \end{cases} \tag{1}$$

函数定义域为 $[t_1,t_k]$，其中 $f_{u,v}(t)$，t_1是从边的起点出发的最早时间，而 t_k 是最后一个可用的出发时间。

FIFO 性质。根据已有研究，[26] [6] [27], 边满足先进先出（FIFO）性质，这意味着如果有两名工作人员在同一条边（u，v）上行驶，那么从 u较早出发的工作人员也会较早到达 v。对于任意 $f_{u,v}(t) \in F$ 和出发时间 $t_1 \le t_2$，我们可以根据这一性质推导出 $t_1+f_{u,v}(t_1) \le t_2+f_{u,v}(t_2)$。

2.2 问题陈述

本小节首先介绍一些基本概念，然后正式给出时变道路网络中共乘插入算子的定义（简称“时变插入”）。

定义2（工作人员）。工作人员表示为 $w=\langle v_w, c_w\rangle$，具有一个已初始化的位置 $v_w \in V$ 和容量 c_w，且其在任意时刻能够搭载的乘客人数 w 不得超过 c_w。

定义3（请求）。一个请求表示为 $r=\langle o_r, d_r, t_r, e_r, c_r\rangle$。该请求在时间 t_r 出现，这意味着该请求的信息直到 t_r 才为人所知。$o_r \in V$ 是该请求的起点，$d_r \in V$ 表示其目的地。如果工作人员在发布时间 t_r 之后于 o_r 接载该请求，并在截止时间 e_r 之前将其送达 d_r，则该请求可以完成。容量 c_r 表示该请求中的乘客人数 r。

在 Uber [4] 和滴滴出行 [1], 等现实世界的共享乘车平台上，工作人员到达 o_r 时接载请求，并在 d_r 将其送达；接载时间和送达时间分别表示为 $arr(o_r)$ 和 $arr(d_r)$。如果工作人员能够服务该请求，则送达时间不得超过请求指定的截止时间即，$arr(d_r) \le e_r$。

定义4（路线）。序列 $\mathcal{S}_R = \langle v_0, v_1, \cdots, v_n \rangle | arr(v_0) = t_0, R = \{r_0, r_1, \cdots, r_{|R|-1}, r_{|R|}\}$ 是该工作人员为服务请求集合 R 而行驶的路线，v_0 表示工作人员当前未送达的请求。w 的当前位置是 v_0，t_0 表示开始时间，w 于时间 t_0 从 v_0 开始行驶这条路线。除 v_0 外，每个顶点都对应于 R 中某个请求的起点或目的地，即 $v_0=v_w$，且对于所有 $1 \le i \le n, v_i \in \{o_r | r \in R\} \cup \{d_r | r \in R\}$。

行程时间。给定一条路线 S_R，其连续顶点为 $v_0, \ldots v_n$，对于从 0 到 n-1 的 k，到达 v_{k+1} 取决于从 v_k出发的时间。如果我们使用数组 $arr[k]$ 记录到达 v_k的时间，那么时间依赖关系可以表示为 $arr[k+1] = arr[k] + disquery(v_k, v_{k+1}, arr[k])$，其中 $disquery(v_k, v_{k+1}, arr[k])$是时变道路网络上的最短旅行时间查询，该函数表示在时间 $arr[k]$出发时从 v_k 到 v_{k+1} 的最短旅行时间。因此，我们可以计算从 t_0 时刻位于路线首个顶点 v_0 出发的整条路线的行程时间，采用递归方式：$arr[1] = t_0 + disquery(v_0, v_1, t_0)$ $arr[2] = arr[1] + disquery(v_1, v_2, arr[1])$ $\ldots$ $arr[n] = arr[n-1] + disquery(v_{n-1}, v_n, arr[n-1])$。

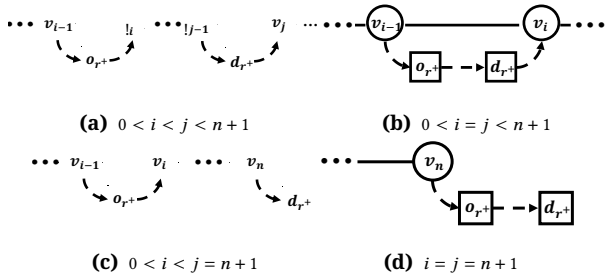


图 2：所有可能配对的插入情况(i, j)。

可行性。如果路线 $S_{\mathcal{R}}$ 满足以下四项约束，我们就称其对工作人员 w 而言是服务请求集合 R的可行路线：

- **完成约束**。工作人员 w 应完成为其分配的所有请求。
- **顺序约束**。$\forall r \in R$, d_r 位于 o_r之后，请求应先由工作人员接载，然后再送至其目的地。
- **截止时间约束**。$\forall r \in R$，工作人员 w 在时间 $arr(d_r)$将 r 送达其目的地 d_r，且送达时间早于其截止时间 e_r。
- **容量约束**。在任何时刻，工作人员 w 已接载但尚未完成服务的请求总数不得超过容量 c_w。

定义 5 (时变插入)。给定工作人员 w 及其关联的可行路线 $\mathcal{S}_{\mathcal{R}}$，该路线从 t_0出发。当新请求r^+在时间 t_{r^+}出现时，算子尝试将 o_{r^+}插入$\mathcal{S}_{\mathcal{R}}$的第 i个位置（即顶点 v_i之前），并将 d_{r^+}插入$\mathcal{S}_{\mathcal{R}}$的第 j个位置（即顶点v_j之前），从而得到新的可行路线 $\mathcal{S}_{\mathcal{R}^+}$，其中$R^+ = R \cup r^+$的行程时间最短。

图2展示了所有可能的插入位置对（i，j），v_0是 w的当前位置，v_n是最后一个顶点，$\forall i, j, 1 \le i \le j \le n+1$。

2.3 基线方法

我们首先将现有的静态道路网络插入算子 [28] 扩展到时变道路网络中，作为算法1所示的基线方法。

基本思路。基线方法背后的直觉如下。(1) 枚举所有可能的位置对（i，j），其中 $1 \le i \le j \le n+1$用于插入新请求的起点 o_{r^+}和新请求的目的地 d_{r^+}，从而得到一条新路线 S_{R^+}；(2) 从属于 S_{R^+}的第一个顶点到最后一个顶点，依次计算行程时间并检查可行性；(3) 如果直到这条新路线的最后一个顶点都没有违反约束，且总行程时间短于当前最优路线的总行程时间，则算法用 S_{R^+}替换最优路线 S^*。

复杂度分析。第2至3行枚举所有潜在的配对 (i, j)，需要 $O(n^2)$ 的时间开销。第5至7行计算插入 o_{r^+}和 d_{r^+}后沿各顶点到最后一个顶点的新的到达时间，需要 $O(n)$ 的时间。第6行涉及在时变道路网络上进行大量行驶时间查询$disquery(v_{k-1}, v_k, arr[k])$；我们假设，与先前研究一样，此类查询的时间开销为 $O(1)$。

工作 [24][28][29]。我们遵循这一假设，尽管该查询的效率不如静态道路网络上的查询函数[26]。因此，算法1的总时间开销为 $O(n^3)$，空间开销为 $O(n)$，用于维护工作人员的路线。

算法1： 基线：立方时间插入

输入： 一名工作人员 w，其可行路线为 S_R，以及一个请求 r^+

输出： w的新路线 S^* 及其旅行成本 obj^*

```
S* ← S_R, obj* ← +∞, arr[0] ← t_0 ;
对于 i ← 1 到 n+1，执行
    对于 j ← i 到 n+1，执行
        S_R+ ← 将 o_r+ 插入到 v_i 之前，并将 d_r+ 插入到 v_j 之前，在 S_R 中;
        对 ⟨v_{k-1}, v_k⟩ ∈ S_R+ 执行
            arr[k] ←
              arr[k-1] + disquery(v_{k-1}, v_k, arr[k-1]) ;
            obj ← arr[k]
        如果 S_R+ 可行 and obj < obj* 则
            S* ← S_R+, obj* ← obj
返回 S*, obj*;
```

示例 *2*。回到示例 *1* 中的设置。对于 r_3，假设我们要检查插入 *(1, 3)* 的可行性并计算总行程时间。根据先前研究，如果将道路网络建模为静态图，我们可以直接得到总行程时间：$arr + dis\ o_1\ o_3 + dis\ o_3\ o_2 + dis\ d_2\ d_3 + dis\ d_3\ d_1$ dis , (d_3) $(,)$ $(,)$ $(,)$ $(,)$，其中 $(\cdot\,\cdot)$ 表示静态图中任意两个顶点之间的最短距离。然而，由于现实中道路路段的行程成本随时间变化，路线中每个顶点的到达时间取决于前一顶点的到达时间。例如，d_2 的到达时间不能直接根据 o_3更新，而是取决于 o_2的新的到达时间。d_1 也是如此，只有知道 d_3 的到达时间后，才能计算d_1 的新的到达时间。因此，如果我们想计算这条新候选路线中每个顶点的到达时间，就需要从o_1开始从头计算。如上一节所示，如果工作人员于 t_1从 o_1 出发，我们按如下方式计算沿途每个顶点的到达时间：$arr(o_3) = arr(o_1) + disquery(o_1, o_3, t_1)$ $arr(o_2) = arr(o_3) + disquery(o_3, o_2, arr(o_3))$ $arr(d_2) = arr(o_2) + disquery(o_2, d_2, arr(o_2))$), $\cdots$ $arr(d_1) = arr(d_3) + disquery(d_3, d_1, arr(d_3))$，$arr(d_1)$ 是这条新路线的总行程时间。对于插入(1, 3)，基线方法调用了 $disquery$ 次函数。

3 我们的方法

为解决基线方案的不足，我们首先提出复合行程时间函数，将行程时间计算和可行性检查加速至 $O(1)$ 时间，从而通过枚举所有 $(i,\ j)$ 值，将时变插入的时间复杂度降低至 $O(n^2)$。随后，我们证明，当沿着工作人员在时变道路网络中的可行路线给定 j 时，我们高效的插入算子可以在 $O(1)$ 时间内找到 i 的最优值。因此，通过沿着工作人员的可行路线枚举 j，时间和空间复杂度均可降低至 $O(n)$。

3.1 二次时间算法

基本思想。插入算子需要 $O(n^2)$ 时间来枚举所有可能的 $(i,\ j)$ 对，以插入新的起讫点对；它只需 $O(1)$ 时间即可计算每个顶点的新的到达时间并检查路线 S_{R^+} 的可行性，而基线方法则需要 $O(n)$ 时间。我们首先将时变道路网络中的边权函数复合为复合行程时间函数，并进一步使用数组 num 记录沿路线已接载但未送达的请求数量。借助复合行程时间函数和 num，检查新路线的可行性并计算其行程时间可加速至 $O(1)$ 时间。为维护当前可行路线 S$\mathcal{R}$中任意两个顶点之间的复合行程时间函数，该方法的空间开销为 $O(n^2)$。

3.1.1 复合旅行时间函数。

为了加快检查和计算，对于一条可行路线 $S_R = v_0, v_1, \ldots v_n$，我们计算复合旅行时间函数$ComTravel(v_x, v_y, t_x)(\forall x, y, 0 < x < y < n)$，以表示从顶点 v_x在时间 t_x出发到顶点 v_y的行程时间。对于每条连续子路线 $v_k, v_{k+1} \in S_R, \forall k \in [0, n-1]$，我们可以得到与之关联的边权重函数 $f_{v_k, v_{k+1}}(t_k) \in F$。如果我们从第一个顶点 v_0 在 t_0出发，将 t_0 与子路线 v_0, v_1 的边权重函数复合，就可以计算出该子路线的复合旅行时间函数 $f_{0,1}(t_0)$；然后递归地将$t_1 = f_{0,1}(t_0)$复合到 $f_{1,2}(t_1)$中，即可计算出子路线 $v_0\ v_1\ v_2$ $travel\ v_0\ v_2\ t_0 = , ,$ $f_{1\,2}\ f_{0\,1}\ t_0 - t_0$的复合旅行时间函数，其中 $(,,)$, $(,())$。按照递归方式执行这一过程，我们给出沿路线的复合旅行时间函数的形式化定义：

定义 *6*（复合旅行时间函数）。给定与 S_R 中子路线 $\langle v_x, v_{x+1}, \cdots v_y\rangle$ 相关联的两个顶点 v_x和 v_y，复合旅行时间函数 $ComTravel(v_x, v_y, t_x)$ 表示路线在时间 t_x从 v_x到 v_y出发时的旅行成本，其中 $ComTravel(v_x, v_y, t_x) = f_{y-1,y}(f_{y-2,y-1}(\cdots f_{x,x+1}(t_x))) - t_x$。

3.1.2 计算到达时间。

After applying 算子 $insert(i, j)$ 中，若将 o_{r^+} 插入 v_i之前，并将 d_{r^+} 插入 v_j之前，则在 S_R中得到一条新的候选路线 S_{R^+}。我们需要计算这条新路线中各顶点的到达时间，以检查其可行性并计算 S_{R^+} 的总行程时间。到达 v_i和 v_j的新的到达时间分别取决于 r^+的接载时间和送达时间，这些时间可通过在道路网络上应用 $disquery$ 进行更新。对于子路线 $v_{i+1}, v_{i+2}, \cdots v_{j-1}$ 中的顶点，其新的到达时间会受到在 v_i之前插入 o_{r^+}的影响；复合旅行时间函数 $ComTravel(v_i, *, *)$ 可用于在 O 时间内更新这些顶点的到达时间，从而避免从头开始递归调用 $disquery$。类似地，对于 $v_{j+1}, v_{j+2}, \cdots v_n$ 中的每个顶点，其新的到达时间取决于到达 v_j 的时间，该时间可通过复合旅行时间函数 $ComTravel(v_j, *, *)$ 计算。

沿着路线 S_R，在第 i 个位置和第 j 个位置插入 o_{r^+} 和 d_{r^+} 时，新请求 $arr(o_{r^+})$ 的接载时间（也就是到达 o_{r^+} 的时间）取决于路线中前一顶点 $arr[i-1]$ 的到达时间。而新请求 $arr(d_{r^+})$ 的送达时间则取决于接载新请求 $arr[j-1]$后前一顶点的新的到达时间

当 $i < j$ 时（见图 2a 和图 2c），或请求 $arr(o_r)$ 的接载时间，当 $i = j$ 时（见图 2b 和图 2d）。

我们首先计算 r^+ 的接载时间。起点 o_{r^+} 插入到 v_i 之后、v_{i-1} 之前，因此可以根据到达 v_{i-1} 的到达时间计算接载时间：

$$arr(o_{r^+}) = arr[i-1] + disquery(v_{i-1}, o_{r^+}, arr[i-1]) \quad (2)$$

类似地，插入 d_{r^+} 后，送达时间取决于接载 r^+ 后到达 v_{j-1} 的新的到达时间，或新请求的接载时间。其中，$arr'[j-1]$ 是在将 o_{r^+} 插入到 v_i 之前后，到达 v_{j-1} 的新的到达时间：

$$arr(d_{r^+}) = \begin{cases} arr'[j-1] + disquery(v_{j-1}, d_{r^+}, arr'[j-1]), & i < j \\ arr(o_{r^+}) + disquery(o_{r^+}, d_{r^+}, arr(o_{r^+})), & i = j \end{cases} \quad (3)$$

根据 S_R 中顶点的位置，我们可以利用复合旅行时间函数，按照以下规则以 (1) 时间计算 O 中每个顶点 v_k 的新的到达时间 $arr'[k]$：

$$arr'[k] = \begin{cases} arr[k], & k < i \\ arr(o_{r^+}) + disquery(o_{r^+}, v_i, arr(o_{r^+}), & k = i \\ arr'[i] + ComTravel(v_i, v_k, arr'[i]), & i < k < j \\ arr(d_{r^+}) + disquery(d_{r^+}, v_j, arr(d_{r^+})), & k = j \\ arr'[j] + ComTravel(v_j, v_k, arr'[j]), & j < k \le n \end{cases} \quad (4)$$

在式（4）中，位于 v_i 之前的顶点不受服务新请求的影响，其到达时间在原路线 S_R 中保持不变。对于顶点 v_i 和 v_j，新的到达时间取决于新请求的接载时间和送达时间；分别调用两个 $disquery$ 函数来计算到达它们的新的到达时间。对于位于 v_{i+1} 和 v_{j-1} 之间的顶点，复合旅行时间函数 $ComTravel(v_i, *, *)$ 可根据到达 v_i 的新的到达时间，计算到达这些顶点的新的到达时间。同样，可以利用复合旅行时间函数 $ComTravel(v_{j,*,*})$ 和到达 v_j 的新的到达时间，计算位于 v_j 之后的顶点的新的到达时间。特别地，最短旅行时间查询和复合旅行时间函数的时间复杂度均为 $O(1)$。因此，对于每个可能的插入位置对 (i, j)，我们可以用 $O(1)$ 时间计算 S_{R^+} 中每个顶点的到达时间。

3.1.3 约束检查与目标计算。

对于一个可能的 (i, j) 插入对，存在两个约束：截止时间约束和容量约束。需要检查这些约束，以确定插入后新生成的路线是否可行。

我们用 $latest[k]$ 表示在不违反 v_k 及其后所有顶点的任何截止时间约束的情况下，到达 v_k 的最晚到达时间。与计算到达顶点的到达时间类似，

$latest[k]$可以根据路线中两个顶点之间的复合旅行时间函数计算得出。对于连续子路线 v_k，$v_{k+1} \in S_R$，为确保工作人员满足从v_k开始的所有顶点的约束，我们需要按逆序计算从 v_n到 v_0的 k的 $latest[k]$。每个顶点的最晚到达时间可通过以下公式计算，而 $latest[n]$ 是

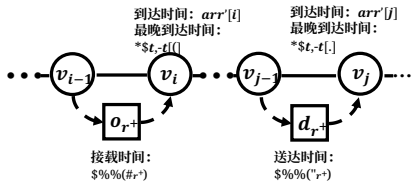


图3：检查截止时间约束。

目的地为最后一个顶点 v_n的请求的截止时间：

$$\begin{cases} latest[k] + ComTravel(v_k, v_{k+1}, latest[k]) = latest[k+1], \\ latest[n] = e_r, \text{ when } v_n = d_r \end{cases} \quad (5)$$

引理1。 当且仅当满足以下条件时，截止时间约束才不会被违反：
(1) $arr(o_{r^+}) \le e_{r^+}$; *(2)* $arr(o_{r^+}) + disquery(o_{r^+}, v_i, arr(o_{r^+}) \le latest[i]$; *(3)* $arr(d_{r^+}) \le e_{r^+}$; *(4)* $arr(d_{r^+}) + disquery(d_{r^+}, v_j, arr(d_{r^+})) \le latest[j]$;

证明。我们按照图3所示，利用一般插入情形证明此引理。对于这个新请求 r^+，我们首先分别计算条件(1)和(3)中的取货时间和送达时间。这两个条件保证这条新的候选路线不会违反 r^+的截止时间约束。条件(2)计算取货后到达 v_i的新的到达时间 r^+, $arr'[i] = arr(o_{r^+}) + disquery(o_{r^+}, v_i, arr(o_{r^+}))$，以确保满足$v_i$后的截止时间约束，$arr'[i]$不得晚于到达 v_i的最晚到达时间。类似地，对于顶点v_j，条件(4)计算服务新请求后到达该顶点的新的到达时间，$disquery\ d_{r^+}\ v_j\ arr\ d_{r^+}$ ，

$[j] = arr(d_{r^+})$ („( )。为避免违反位于v_j之后的顶点的截止时间约束，条件(4)保证 $arr'[j]$早于到达v_j的最晚到达时间。□

为了在 $O(1)$ 时间内检查容量约束，我们沿用先前研究 [24]，并用 $num[k]$ 表示在 v_k已接客但尚未送达的请求数量。然后按照以下规则计算该数组，其中 v_k 是请求 $r \in R$的接客地点 o_r或送达地点 d_r：

$$num[k] = \begin{cases} num[k-1] + c_r, & v_k = o_r \\ num[k-1] - c_r, & v_k = d_r \end{cases} \quad (6)$$

引理2。 *[24]* 当且仅当 *(*
1 $num[i-1] \le c_w - c_{r^+}$ *2* $num[k] \le$*) ; ( ) , .,* 新路线上 $c_w - c_{r^+}$ $i < k \le j$
的容量约束不会被违反。

引理2的证明请参见参考文献[24]。

3.1.4 算法细节。

主要思想。算法枚举所有可能的（i, j）对，并在 $O(1)$ 时间内根据方程（2）至方程（4）更新新的到达时间。对于 v_i，利用引理1（1）–（2）检查插入 o_{r^+}后是否会违反截止时间约束，并通过引理2（1）检查容量约束。利用引理1（3）–（4）和引理2（2），检查在 v_j之前插入 d_{r^+}后是否违反截止时间约束和容量约束。所有这些可行性检查的时间开销为 $O(1)$。

复杂度分析。 第 2 行初始化 arr，$latest$，num。第 3 行复合任意两个顶点之间的旅行时间函数

算法 2： 二次时间插入

输入： 一名工作人员 w，其路线可行 S_R，以及一个请求 r^+
输出： 新路线 S^*，对应 w，以及其旅行成本 obj^*

$S^* \leftarrow S_R, obj^* \leftarrow +\infty$;
通过 *Eq.* (4) - *Eq.* (6) 初始化 $arr, latest, num$;
复合行程时间函数 $ComTravel(i, j, \cdot)\ \forall i, j, 0 < i < j < n$ 根据定义6;
对于 $i \leftarrow 1$ 到 $n+1$ **执行**
  若 引理*1*的*(1)*或*(2)*不成立**则**继续;
  若 引理*2*的*(1)*不成立**则** 继续;
  对 $j \leftarrow i$ 至 $n+1$ **执行**
    若 引理*1 (3)* 或 *(4)* 被违反**则**继续;
    若 引理*2 (2)* 被违反**则** 跳出;
    若 $j = n+1$**则**
      $obj \leftarrow arr(d_{r^+})$
    else
      根据 *Eq.* (4) 计算 $arr'[j]$;
      $obj \leftarrow arr'[j] + ComTravel(v_j, v_n, arr'[j])$;
    如果 $obj < obj^*$ **则**
      $obj^* \leftarrow obj,\ i^* \leftarrow i,\ j^* \leftarrow j$
  如果 $obj^* < +\infty$ **则**
    将 $S^* \leftarrow$ 插入 S_R 中的第 i^* 个元素之前，并将 o_r 插入第 j^* 个元素之前
返回 $S^*,\ obj^*$;

表 2：算法 2 中 S_R 的 *ComTravel*

	o_1	o_2	d_2	d_1
o_1	×	$ComTravel(o_1, o_2, t)$	$ComTravel(o_1, d_2, t)$	$ComTravel(o_1, d_1, t)$
o_2	×	×	$ComTravel(o_2, d_2, t)$	$ComTravel(o_2, d_1, t)$
d_2	×	×	×	$ComTravel(d_2, d_1, t)$
d_1	×	×	×	×

沿着 S_R。第 4 行和第 7 行分别枚举插入 o_{r^+} 和 d_{r^+} 的可能位置，每行耗时 $O(n)$。第 5 行和第 6 行检查插入 o_{r^+} 时是否满足截止期限和容量约束。对于可能的第 j 个位置，第 8 行和第 9 行检查截止期限和容量约束。每行约束检查耗时 $O(1)$。如果第 10 行的 $j = n+1$，如图 2d 和图 2c 所示，r^+ 是最后一个已配送请求，则新路线的总行程时间就是新请求的送达时间。否则，根据第 13 行的方程 4，我们可以在 $O(1)$ 时间内插入新请求的起点和目的地后，得到 v_j 处新的到达时间；然后第 14 行耗时 $O(1)$，基于复合旅行时间函数和 v_j 处新的到达时间，计算从 v_j 到 v_n 的目标函数。因此，算法 2 的总时间开销为 $O(n^2)$。对于空间开销，第 3 行需要初始化任意两个顶点之间的一个复合旅行时间函数，因此一个工作人员需要保存 $O(n^2)$ 个旅行时间函数，空间开销为 $O(n^2)$。

示例 3. 回到示例 *1* 中的设置。对于行程时间最短的可行路线 $S_R = \langle o_1, o_2, d_2, d_1 \rangle$，沿着 S_R 的复合旅行时间函数如表 *2* 所示。对于 S_R 中的每个顶点，算法都会复合并维护从该顶点到其后所有顶点的行程时间函数。因此，对于每个顶点，我们可以利用这些函数更新到其后任意顶点的到达时间，而不必

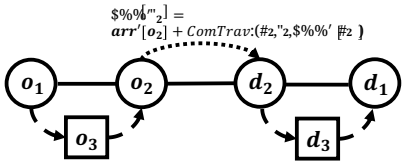


图 4：使用 *ComTravel* 更新到达时间。

在大规模时变道路网络上运行最短路径查询算法。对于 r_3，我们还考虑插入(1, 3)。如图 *4* 所示，在 o_2 之前插入 o_3 后，与前一个示例相同，必须调用 *2 disquery* 函数来计算 $arr'[1]$。然而，由于 o_2 维护了一个到其后 d_2 的复合旅行时间函数，因此可以直接将到 d_2 的新的到达时间更新为 $arr'[1] + ComTravel(o_2, d_2, arr'[1])$，如图 *4* 中的点线所示。要计算最终行程时间 $arr'[2]$，需基于更新后的 $arr'[2]$，从 d_2 调用 *2 disquery*。因此，算法 *2* 复合 *3* 个行程时间函数；对于插入(1,3)，需要调用 *disquery* 次函数。

3.2 线性时间算法

基本思想。 不枚举插入新请求起点和目的地的可能位置，而是维护总计所有 $O(n^2)$ 个 $(i,\ j)$ 对的复合行程时间函数，我们枚举 j 的值并找出最优 i。然后，我们进一步基于复合行程时间函数，在 $O(1)$ 时间内计算新的行程时间。因此，工作人员只需存储从 v_j 到最后一个顶点 v_n 以及到 v_j 的下一个顶点的复合行程时间函数，其中 j 是当前路线中插入新请求目的地的每个可能的 j 位置，复合行程时间函数的空间开销可从 $O(n^2)$ 降低到 $O(n)$。

3.2.1 枚举 j 时更新最优接载位置。
给定一个顶点 v_k 及其路线 S_R，如果将新请求 d_{r^+} 的目的地插入到该顶点之前，根据顺序约束，插入接载顶点 o_{r^+} 只有两种情况。情况1（$i = j = k$），如图 2b 和图 2d 所示，起点也插入到当前路线中的 v_k 之前。情况2（$i < j$），如图 2a 和图 2c 所示，目的地插入到 v_k 之前，而起点 o_{r^+} 插入到子路线 v_1，v_2、$\cdots$ v_{k-1} 中的 v_k 之前。

基于时变道路网络的FIFO性质，如果我们能够更早到达顶点 v_k，那么就能更早到达最后一个顶点 v_n，从而得到一条总行程时间更短的更优路线。然后，在枚举 j 的值时，我们可以更新插入起点的最优位置；在将 d_{r^+} 插入 v_j 之前后，我们进一步检查：在子路线 v_1，v_2，$\cdots$ v_{j-1} 中，将 o_{r^+} 插入 v_j 之前是否比将 o_{r^+} 插入更优。我们使用 $pckl[k]$ 表示在将 d_{r^+} 插入 v_k 之前插入 o_{r^+} 的最佳位置；如果这个新的潜在第 k 个位置不满足插入 o_{r^+} 时的截止时间约束和容量约束中的任一项，则 $pckl[k]$ = 空；否则，我们可以基于以下引理计算其值：

引理3. 给定一个待插入顶点 v_k，将其插入到该顶点之前时，当且仅当满足以下条件，我们才能更新 $pckl[k] = k$：*(1)* $num[k-1] \le c_w - c_{r^+}$; *(2)* $arr(o_{r^+})$ <= e_{r^+} 和 $arr(o_{r^+}) + disquery(o_{r^+}, v_k, arr(o_{r^+})) \le$

$latest[k]$; *(3)* $arr(o_{r^+}) + disquery(o_{r^+}, v_k, arr(o_{r^+})) < arr'_{o_{r^+}}[k]$，其中 $arr'_{o_{r^+}}[k]$ 是将 o_{r^+} 插入子路线 $\langle v_1, v_2, \cdots v_{k-1}\rangle$ 的相应位置后到达 v_k 的到达时间，可根据从 v_{k-1} 到 v_k 的旅行时间函数计算得出

$$arr'_{o_{r^+}}[k] = arr'_{o_{r^+}}[k-1] + ComTravel(v_{k-1}, v_k, arr'_{o_{r^+}}[k-1])$$

证明。条件（1）检查容量约束，确保在 v_k 处工作人员有足够的容量来取货 r^+。条件（2）检查在 o_{r^+} 之前插入 v_k时是否满足截止时间约束。条件（3）保证在新的潜在第 k 个位置插入 o_{r^+} 所需的行程时间，比将其插入子路线 v_1，v_2，$\cdots$ v_{k-1}更短；我们可以用反证法证明这一点。对于新的潜在位置 k，如果将 o_{r^+} 插入其之前，到达的新的时间等于 $arr(o_{r^+}) + disquery(o_{r^+}, v_k, arr(o_{r^+}))$。在考虑 r^+ 之前，取送新请求 r^+ 的最佳位置是 $pckl[k-1]$；如果将 o_{r^+} 插入 $pckl[k-1]$处，我们可以根据从 v_{k-1} 出发的到达时间 $arr'_{o_{r^+}}[k]$，复合旅行时间函数，得到到达 v_k的时间，其中 $arr'_{o_{r^+}}[k-1] + ComTravel(v_{k-1}, v_k, arr'_{o_{r^+}}[k-1])$。如果条件（3）成立，我们就能更早到达 v_k。假设$pckl[k-1]$是插入 o_{r^+}的最佳位置，那么与将 o_{r^+}插入 v_k之前相比，我们可以得到一条行程时间更短的路线。然而，根据 FIFO 性质，如果选择 $pckl[k-1]$，那么到达 v_k 以及最终顶点 v_n 的时间会更晚，行程时间也会更长，这与假设相矛盾。该假设不成立，因此 k是插入 o_r的更好位置，我们可以更新 $pckl[k] = k$。□

3.2.2 在 $pckl[k]$的 o_{r^+} 处插入新请求的起点。

枚举每个潜在的第 k 个位置时，根据引理3，我们可以在 $O(1)$ 时间内更新插入新请求起点的最佳位置。为了进一步检查插入 d_{r^+} 的可行性，我们需要先根据 $pckl[k]$ 的值更新到达 v_k 的新到达时间。如果该值为 空，则在当前路线中，直到顶点 v_k 都没有可用于接载和送达 r^+ 的可行位置，因此到达时间保持原值。如果 k 是插入 o_{r^+} 的最优位置，即 $pckl[k] = k$，也就是将 o_{r^+} 插入 v_k 之前，则会调用两次最短旅行时间查询 $disquery$，以获取从 v_{k-1} 到 o_{r^+} 的接载时间，以及从 o_{r^+} 到 v_k 的新到达时间。否则，$pckl[k]$ 的值表示在 v_k 之前的子路线中插入起点的最佳位置。可以根据复合旅行时间函数以及从前一顶点 v_{k-1} 出发的到达时间，更新到达 v_k 的新到达时间。

如果空，我们可以根据以下规则，在 $pckl[k] \neq k-1$ 时间更新 v_k的新的到达时间。情况表示，将 o_{r^+} 插入到之前。情况表示，将插入到子路线 $v_1 v_2 \cdots v_{k-1}$ 中：

$$arr'_{o_{r^+}}[k] = \begin{cases} arr(o_{r^+}) + disquery(o_{r^+}, v_k, arr(o_{r^+})), & Case1 \\ arr'_{o_{r^+}}[k-1] + ComTravel(v_{k-1}, v_k, arr'_{o_{r^+}}[k-1]), & Case2 \end{cases} \quad (7)$$

3.2.3 在第 k个位置插入新请求的 d_{r^+}。

对于每个第 k个位置，我们应考虑工作人员的容量约束、新请求的时间约束以及在 v_k处的新的到达时间，以检查是否可以在 v_k之前插入 d_{r^+}。

表3：算法3中 S_R 的 $ComTravel$

	1	2
o_1	$ComTravel(o_1, o_2, t)$	$ComTravel(o_1, d_1, t)$
o_2	$ComTravel(o_2, d_2, t)$	$ComTravel(o_2, d_1, t)$
d_2	$ComTravel(d_2, d_1, t)$	×
d_1	×	×

引理4。 k号位置是一个可行位置，可用于插入 d_{r^+}，当且仅当 *(1)* $num[k-1] \leq c_w - c_{r^+}$; *(2)* $arr(d_{r^+}) \leq e_{r^+}$; *(3)*$arr(d_{r^+}) + disquery(d_{r^+}, v_k, arr(d_{r^+})) \leq latest[k]$

证明。条件 (1) 检查容量约束。条件 (2) 确保在第 k 个位置送达新请求不会违反新请求的截止时间约束。在 v_k 之前插入d_{r^+}后，条件 (3) 检查到 v_k 之后送达的已分配请求中，前往 v_k 的新的到达时间不会违反任何截止时间约束。□

如果将 d_{r^+} 插入 k 是可行的，我们首先更新受插入 o_{r^+} 影响的 v_{k-1} 处的新到达时间。接下来，我们可以得到新请求的送达时间 $arr(d_{r^+})$：

$$arr(d_{r^+}) = arr'_{o_{r^+}}[k-1] + disquery(v_{k-1}, d_{r^+}, arr'_{o_{r^+}}[k-1]) \quad (8)$$

然后，我们可以根据查询函数 $disquery$ 更新送达新请求后到达 v_k 的新的到达时间。我们使用 Δ_k 表示在 $pckl[k]$ 处插入 o_{r^+} 并在 k 处插入 d_{r^+} 时到达 v_k 的时间，

$$\Delta_k = arr(d_{r^+}) + disquery(d_{r^+}, v_k, arr(d_{r^+})) \quad (9)$$

由于 v_k 维护了直到最后一个顶点 v_n 的复合旅行时间函数，因此我们可以计算到达 v_n 的新的到达时间，该时间是插入（$pckl[k]$, k）的 $objective$，

$$obj = \Delta_k + ComTravel(v_k, v_n, \Delta_k) \quad (10)$$

3.2.4 算法细节。

主要思想。 我们枚举当前路线中从 v_0 到 v_n 的每个第 k 个位置，并更新 $pckl[k]$ 值，以指示在当前路线中插入 r^+ 的起点直到 v_k 的最佳位置。同时，对于每个 k，我们通过在 $pckl[k]$ 处插入 r^+ 的起点并在 k 位置插入目的地，检查可行性并计算由此生成的新路线的目标函数。对于这条候选路线，其中在 $pckl[k]$ 处接载 r^+ 并在 v_k 之前送达，通过引理4检查可行性，并根据公式（9）和公式（10）计算新的总行程时间目标函数。找到插入新请求起点的最优位置、检查可行性并计算这条新路线的目标函数需要 $O(1)$ 的时间。

复杂度分析。 第2行初始化 arr、$latest$和 num。第3行计算 v_k到 v_{k+1} 以及 v_k到v_n 之间的旅行时间函数，耗时 $v_k \in S_R$。第4行枚举插入 r^+的目的地的每个潜在位置，耗时 $O(n)$。第5行检查在新的潜在第 k 个位置插入 o_{r^+} 时的截止期限和容量约束；如果违反约束，则插入新请求起点的最佳位置位于子路线 v_1、v_2、……、v_{k-1} 中。否则，第6行中的 k 将是插入起点的最佳位置。插入起点后，需要在第7行更新受其影响的到达时间。接着，在第8—10行中检查可行性，并计算将 d_{r^+} 插入 v_k 之前后到达 v_k 的新的到达时间，每行耗时 $O(1)$。插入新请求的起点和目的地后，根据第11—14行计算新路线的行程时间。因此，算法3的总时间开销为 $O(n)$。至于空间

算法3： 线性时间插入

输入： 一名工作人员 w，其路线可行 S_R，以及一个请求 r^+

输出： 新路线 S^*，适用于 w，其旅行成本为 obj^*

$S^* \leftarrow S_R, obj^* \leftarrow +\infty$;

通过 *Eq.* (4) - *Eq.* (6) 初始化 $arr, latest, num$;

复合旅行时间函数 $ComTravel(k, k+1, \cdot)$ $ComTravel(k, n, \cdot)$ $\forall k, 0 < k < n$通过定义 6 ;

对于 $k \leftarrow 1$ 至 $n+1$，**执行**

  如果 引理*3 (1)* 或 *(2)* 不成立 **则**继续;

  如果 引理*3 (3)* **那么** $pckl[k] \leftarrow k$;

  更新 $arr'_{o_r^+}[k]$ 插入后o_r^+ 基于方程 (7);

  如果引理*4 (1)(2)(3)* **则**

    基于插入 o_r^+ 后计算 $arr(d_r^+)$ 公式（8）;

    根据以下内容计算插入的 Δ_k（$pckl[k]$, k）公式（9）;

    如果 $k = n+1$, **则**

      $obj \leftarrow \Delta_k$;

    else

      $obj \leftarrow calculateby$ 公式（10）;

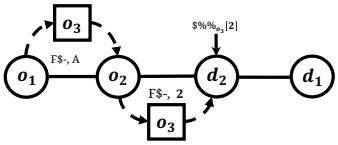
    如果 $obj < obj^*$，**则**

      $obj^* \leftarrow obj, i^* \leftarrow pckl[k], j^* \leftarrow k$;

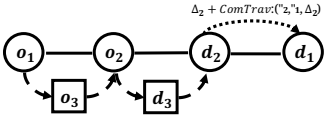
如果 $obj^* < +\infty$，**则**

  S^* ← 将 o_r 插入到 S_R 中的第 i^* 个位置，并将 d_r 插入到第 j^* 个位置;

返回 S^*, obj^*;



(a) 当 $k = 2$**时，更新** $pckl[2]$



(b) 插入的 $obj(pckl[2], 2)$

图5：插入$(pckl[2], 2)$。

开销，对于第2行中的每个顶点 v_k，需要初始化两个复合旅行时间函数，因此一个工作人员应维护 $O(n)$ 个复合旅行时间函数，空间开销为 $O(n)$。

示例4。 回到示例 *3* 中的设置。对于行程时间最短的可行路线 $S_R = o_1, o_2, d_2, d_1$，沿着 S_R的复合旅行时间函数如表*3*所示。每个顶点最多组合并维护 *2* 个函数，一个对应下一个顶点，另一个对应 S_R中的最后一个顶点。为便于说明，我们首先假设插入 (1，2) 是服务 r_3的最优位置对。当 $k = 1$时，表示 d_3 插入在 o_2 之前。由于顺序约束， o_3 只能插入在 o_2之前。我们首先计算 o_2 处的新的到达时间。如果 o_3 插入在其之前，则 $arr'_{o_3}[1] = arr(o_3) +$ $disquery(o_3, o_2, arr(o_3))$ 和 $pckl[1] = 1$。当 $k = 2$ 时，如图 *5a* 所示，在 d_2 之前出现了一个可插入 o_3 的新位置。由于 *FIFO*性质，如果我们能在 d_2 处获得更早的到达时间，就能在 d_1 处获得更早的到达时间。因此，当 $k = 2$ 时，我们需要检查哪个位置最适合插入 o_3，即 $pckl[2] = 2$ 还是 *1*。在情况 *1* 中， o_3 插入在 o_2 之前， d_2 处的新的到达时间可计算为 $arr'_{o_3}[1] + ComTravel(o_2, d_2, arr'_{o_3}[1])$。否则，在情况*2*中， o_3 插入在 o_2 之前， d_2 处的新的到达时间可计算为 $arr(o_3) + disquery(o_3, d_2, arr(o_3))$。根据我们的假设， $arr'_{o_3}[1] + ComTravel(o_2, d_2, arr'_{o_3}[1])$ 小于 $arr\ o_3 + disquery\ o_3\ d_2\ arr\ o_3$ $arr_{o_3}[2] = arr_{o_3}[1] + ComTravel\ o_2\ d_2\ arr_{o_3}[1]$ $pckl[2] = 1$，因此 $arr'_{o_3}[1] + disquery(o_2, o_3, arr'_{o_3}[1])$。然后我们尝试将 d_3 插入在 d_2 之前，首先可得到 $arr(d_3)$，其值为 $arr'_{o_3}[1] + disquery(o_2, o_3, arr'_{o_3}[1])$。考虑图 *5b* 中将 o_3 插入在 o_2 之前，并将 d_3 插入在 d_2 之前的情况，即插入 *(1, 2)*。我们可以得到 d_2 处的到达时间为 $\Delta_2 = arr(d_3) + disquery(d_3, d_2, arr(d_3))$。最后，根据从 d_2 到最后一个顶点的旅行时间函数，我们可以得到目标值 $\Delta_2 + ComTravel(d_2, d_1, \Delta_2)$。

表4：数据集统计信息。

数据集	#(顶点)	#(边)	#(插值点)
成都	423，434	913，718	1，411，569
海口	41, 542	89, 206	138, 083

4 实验研究

本节首先介绍本研究的实验设置，然后展示并讨论实验结果。

4.1 实验设置

数据集。 我们在两个真实世界数据集上评估所提出的方法。第一个是从中国成都市收集的公开数据集，由滴滴出行通过 GAIA [2] 计划发布。第二个数据集收集自中国海口市。我们从 OpenStreetMap [3], 下载这两座城市的道路网络，然后按照先前研究 [26],中的设置和流程生成时变道路网络。数据集中的边数从 80,000 条到 900,000 多条不等。我们将时间域设为 86,400 秒，即全天。对于时变道路网络中的边，相关时变权重函数的插值点数量分别为 138,083 和 1,411,569。我们使用 [26] 中的 TDSP 查询函数进行最短旅行时间查询。数据集的统计信息见表4。

我们选择白天请求数量最多的日期数据进行评估，白天时间为即上午 8:00 至下午 18:00。两个数据集分别表示为成都和海口 。数据集中的每个请求都是一个由起点、目的地和发布时间组成的元组。请求的起点和目的地是位于城市内的纬度和经度坐标对，我们将请求的起点和目的地映射到城市道路网络中最近的顶点。对于每个请求，我们将从发布时间到截止期限的时间段设为 10 至 30 分钟。在时间段 $e_r - t_r$ 内，该请求可以与其他已分配请求一起由拼车服务提供服务。工作人员的起始顶点从道路网络的顶点中随机选取，每位工作人员的容量在 3 至 20 之间变化。表 6 总结了本实验的主要参数，这些参数也用于已有研究 [14], [20],

表5：对比算法。

算法	时间复杂度	空间复杂度
三次时间算法	$O(n^3)$	$O(n)$
二次时间算法	$O(n^2)$	$O(n^2)$
线性时间算法	$O(n)$	$O(n)$

[31], [22], [30], [29]。最近，一种名为“巴士拼车”的热门拼车服务吸引了大量关注 [5], ，这是一种可描述为“巴士版 Uber”的交通解决方案。因此，我们将工作人员的最大容量设为 20。

对比算法。本节对比以下算法。表5总结了这些算法。

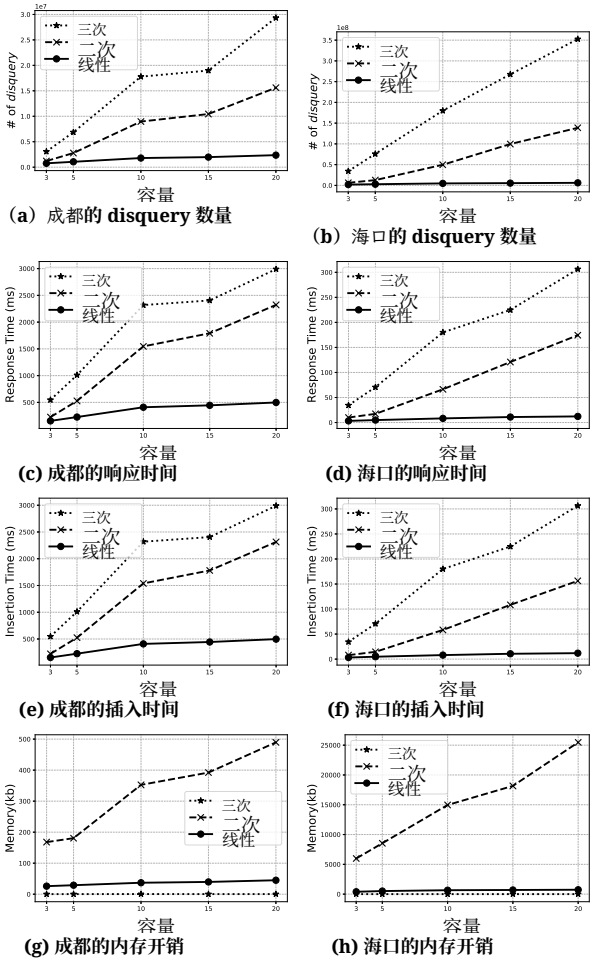
- 三次时间算法（算法 1）。 本文的插入算子。 它枚举所有可能的 $(i,\ j)$ 对，以生成新的候选路线。对于每条路线，递归调用时变道路网络上的最短旅行时间查询，以计算到达时间并检查该路线是否满足约束。就空间开销而言，工作人员只需维护当前的可行路线。
- 二次时间算法（本文的算法 2）。该方法通过枚举所有可能的 $(i,\ j)$ 对来生成新的候选路线。为降低时间复杂度并避免调用大量最短旅行时间查询，工作人员将时变边权重复合起来，并维护复合旅行时间函数。除辅助数组外，空间开销主要取决于工作人员可行路线中任意一对顶点之间的复合旅行时间函数，因此空间开销为 $O(n^2)$。
- 线性时间算法（算法 3）。 该方法不再枚举这两个本文的值，而是枚举 j，并在 O 时间内找到最优的 i，耗时为 $O(1)$。最短旅行时间查询的调用次数也进一步减少。时间和空间开销均降至 $O(n)$，因为该方法会维护从可行路线上的每个顶点到其下一个顶点和最后一个顶点的复合旅行时间函数，而不是检查所有顶点对。

指标。以下指标对于评估时变插入的性能至关重要：(1) 工作人员容量c_w的影响；(2) 每个请求的时间段 $e_r - t_r$的影响；(3) 请求数量的影响；(4) 工作人员数量对可扩展性的影响。我们选择需要评估的指标：(1) 最短旅行时间查询的调用次数

时间查询 $disquery$；（2）插入时间；（3）响应时间；（4）内存开销。时变道路网络上的最短旅行时间查询是时变插入的瓶颈，因此我们避免调用大量 $disquery$ 以提高效率。对于插入时间这一指标，在三次时间算法 和二次时间算法中，它表示沿当前路线检查可行性并计算每个可能的（$i,\ j$）对的目标函数所需的平均时间。在线性时间算法中，它表示针对每个 j 值检查可行性，然后沿当前路线寻找最优 i 并计算目标函数所需的平均时间。响应时间和内存开销都是许多拼车应用中的指标 [16][14][24]。响应时间是为每个请求分配工作人员所需的平均时间。在三次时间算法中，内存开销是用于存储工作人员路线的内存。内存开销

表 6：参数设置。

参数	设置
容量 c_w	3，5，10，15，20
时间段：$e_r - t_r$（分钟）	10，15，20，25，30
请求数量	20k，40k，60k，80k，100k
可扩展性：工作人员数量	100，200，300，400，500



（a）成都的 disquery 数量
（b）海口的 disquery 数量
(c) 成都的响应时间
(d) 海口的响应时间
(e) 成都的插入时间
(f) 海口的插入时间
(g) 成都的内存开销
(h) 海口的内存开销

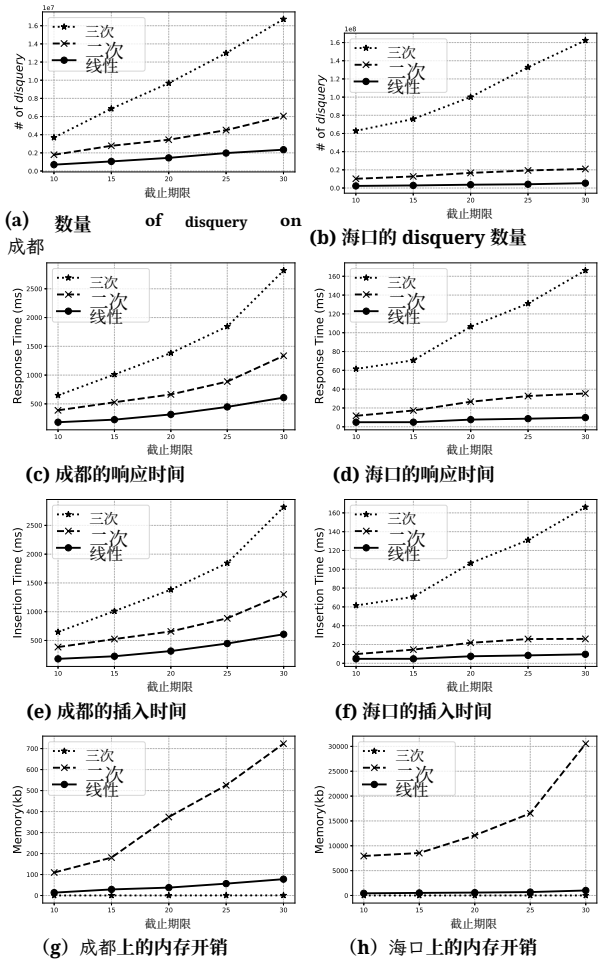
图 6：工作人员容量变化的结果

二次时间算法和线性时间算法的内存占用均主要由工作人员维护的复合行程时间函数决定，工作人员还会存储路线和辅助数组。

实现。 实验在一台服务器上进行，该服务器配备 40 个支持超线程的 Intel(R) Xeon(R) E5 2.30GHz 处理器和 128GB 内存。三种提出的插入算子均使用 GNU C++ 实现。按照 [11, 12, 19], 中的设置，每项实验重复进行 10 次，并报告平均结果。

4.2 实验结果

工作者容量的影响。 图 6 展示了在成都 和海口改变工作者容量后的结果。线性时间算法 调用最短行程



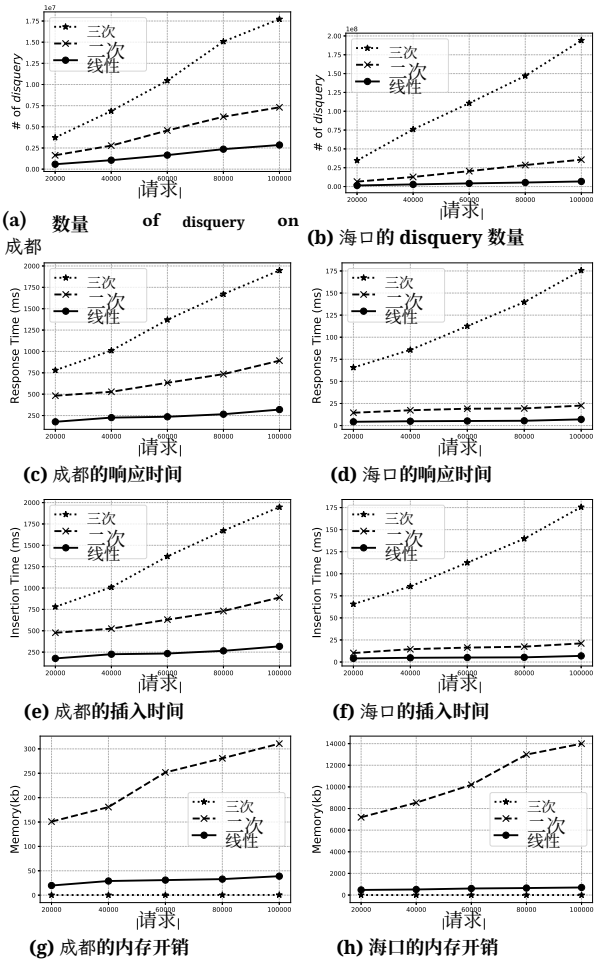
(a) 数量 of disquery on 成都 (b) 海口的 disquery 数量

(c) 成都的响应时间 (d) 海口的响应时间

(e) 成都的插入时间 (f) 海口的插入时间

（g）成都上的内存开销 （h）海口上的内存开销

图7：改变 $e_r - t_r$的结果



(a) 数量 of disquery on 成都 (b) 海口的 disquery 数量

(c) 成都的响应时间 (d) 海口的响应时间

(e) 成都的插入时间 (f) 海口的插入时间

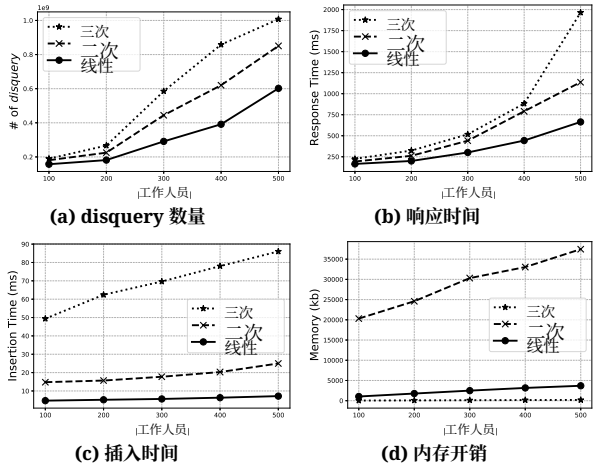
(g) 成都的内存开销 (h) 海口的内存开销

图8：请求数量变化时的结果

时间查询的次数最少，相较于基线方法。线性时间算法 在这两个数据集上将调用 *disquery* 的次数减少了 75.24% - 91.94%。调用 *disquery* 次数的大幅减少证明了复合旅行时间函数的有效性。相应地，线性时间算法 的插入时间最快，在成都上比三次时间算法快至 6.01 倍，在海口 上快至 25.79 倍。我们可以观察到，二次时间算法 的内存开销远高于另外两种方法，这是因为时变插入的内存开销主要由复合旅行时间函数决定。与二次时间算法相比，线性时间算法 在成都 和海口分别减少了 90.8% 和 97.1% 的内存开销。随着工作者容量增加，时间开销和内存开销呈现相同的增长趋势，而线性时间算法 始终耗时最少，且内存开销远低于二次时间算法（在成都上不超过 45 KB）。值得注意的是，三次时间算法的插入时间和响应时间，以及二次时间算法 的内存开销，在成都 和海口 均发生显著变化，这是由于 $O(n^3)$ 时间复杂度和 $O(n^2)$ 空间复杂度所致。

截止期限的影响 e_r。图7展示了在成都和海口上请求截止期限变化时的结果。 $e_r - t_r$的值表示 x 轴的刻度值。同样，在时间开销方面，线性时间算法优于另外两种算法；在响应时间上，它在成都快达 4.62 倍，在海口快达 16.96 倍。在空间开销方面，它仍然比二次时间算法具有显著优势。随着 $e_r - t_r$增加，请求的截止期限变长，更多请求可以插入原有的可行路线，三次时间算法的时间开销和二次时间算法的空间开销显著增加。同时，线性时间算法调用 *disquery* 函数的次数、插入时间和响应时间的增长速度都远低于另外两种方法。在内存开销方面，随着请求截止期限增加，线性时间算法的开销仍相对稳定。与空间开销为 $O(n)$ 的基线方法相比，线性时间算法仅消耗略多的内存（不超过 77 KB，属于可接受范围）。

请求数量影响。图 8 展示了在成都和海口改变请求数量后的结果。线性时间算法仍是三种算法中最快的，即，在两个数据集上，其请求数量分别比基线少 6.21 倍和 25.71 倍，响应时间分别快 6.09 倍和 12.22 倍。



(a) disquery 数量 **(b) 响应时间**

(c) 插入时间 **(d) 内存开销**

图 9：海口可扩展性测试结果

对于成都和海口，随着请求数量增加，工作人员需要获取行程时间更长的路线来服务请求，因此三种算法的时间开销都会增加。至于内存开销，二次时间算法由于空间复杂度较高，表现仍然最差，尽管其运行速度至少比三次时间算法快 4.59 倍。随着两个数据集中请求数量的增加，线性时间算法的内存开销略有增加。

可扩展性测试。本实验验证了时变插入的可扩展性。我们在图 9 中展示了海口 的结果，而成都 的结果也呈现出相同趋势。对于三次时间算法，随着工作者数量增加，调用次数 *disquery* 呈指数增长，导致响应时间显著变慢。结果证明，现有插入算子在时变道路网络中可能成为瓶颈，尤其是在成都这样规模更大的城市中。对于二次时间算法，工作者数量较多时，内存开销很容易变得显著。除内存开销外，线性时间算法 在所有指标上均优于另外两种方法，并且能够实时分配请求。与三次时间算法相比，线性时间算法 的 $O(n)$ 空间复杂度也可以接受（小于 3.7 MB）。结果表明，我们提出的方法适用于真实的大规模道路网络。

数据集比较。 通过比较成都 和海口上的结果，我们得出以下观察结论：

- 线性时间算法 在两个数据集上的运行速度都极快，该方法可以实时处理请求。与海口相比，时变插入算子在成都上的运行速度较慢，这是因为最短行程时间查询 *disquery* 成为了更大道路网络中的瓶颈。
- 就内存开销而言，三种方法在海口 上消耗的内存（ 0.496 - 30541 KB）均高于成都 （0.238 - 723 KB）。这可能是因为海口 的请求更有可能共享一条路线，时变插入算子为每个请求的起点—目的地对提供了更多可行插入位置，从而增加了内存开销。

实验总结。我们总结结果如下：

- 将现有的插入算子直接扩展到时变道路网络上的拼车服务并不可行。换句话说，基线方法在成都上的响应时间超过 2990 毫秒，无法实时处理请求。

- 由于时间复杂度得到了改进，我们的优化算法二次时间算法和线性时间算法 在成都上分别比基线快 2.16 倍和 6.09 倍；至于海口 ，这两种算法分别比基线快 6.39 至 25.79 倍。• 线性时间算法 相比二次时间算法最多可将内存开销降低 97.1%。线性时间算法 消耗的内存仅略高于三次时间算法 ，例如，在成都上不超过 77 KB。

5 相关工作

本节讨论相关工作。具体而言，相关工作可分为两类：时变道路网络上的路线处理，以及拼车路径规划。

时变道路网络上的路线处理。时变道路网络在空间数据库领域吸引了许多研究关注。分段线性函数被广泛用于拟合时变边权函数，以模拟现实中的动态交通 [26] [6] [27]。针对时变道路网络上的基础查询问题， [10] 提出了一种新颖的双层路径索引；给定路径规划查询时，会利用基于该索引的过滤与细化策略来提高路径规划的效率。 [26] 将道路网络划分为层次分区，并构建了平衡树索引。遇到查询时，该方法会组合分区中边界顶点的分段线性函数来回答查询。

时变道路网络上的路径规划相关问题也至关重要。 [6] 提出了时变道路网络上的在线路径规划问题，并开发了一种复杂度为 $O(|R|m + |R|nlogn)$ 的请求插入算法，以降低竞争比。 [9], 将最后一公里配送问题扩展到时变道路网络：快递员可以从同一个仓库出发，携带多个包裹；每个包裹可根据时间配送至不同的备选地点。为最大化已配送包裹的数量，该文提出了一种优化的插入启发式方法，将配送地点插入快递员的路线。然而，现有工作每个请求只处理一个位置；[6] 假设所有乘客具有相同的目的地，而 [9] 假设所有包裹具有相同的起点。此外， [6] 和 [9] 都没有检查工作人员的可行性。与这些问题相比，我们的时变插入问题要困难得多。如示例 1 所示，工作人员 w从每个请求各自的起点接载乘客，并将其送至各自的目的地。对于新请求 r_3，我们规划一条新的可行路线 o_1， o_2， o_3， d_2， d_3， d_1，其中包含其起点 o_3和目的地 d_3，以接载并送达该乘客。因此，在不假设请求具有相同起点或目的地的情况下，现有算法无法应用于我们的问题。

拼车路径规划。拼车服务也在许多领域得到研究，涵盖数据库和人工智能。不同研究关注不同的目标函数， [16][15] 致力于最大化

已服务的请求数量，[13][18][14][7][28] 则关注另一个重要目标函数，即最小化总行程时间，这与我们的优化目标相似。从平台的角度来看，总收入也是需要考虑的目标函数 [32][33]。

插入算子是一种高效的方法，可根据优化后的目标函数，将新请求插入工作人员的当前路线，从而规划一条新的可行路线，相关方法见 [20]。近期，也有一些研究证明了该方法在大规模真实世界静态道路网络中的有效性和效率 [24][28][29]. [28] [29]对插入算子进行了深入研究，提出了一种基于动态规划的新型划分框架，将时间复杂度从 $O(n^3)$ 降至 $O(n^2)$，随后利用芬威克树索引进一步将复杂度降至 $O(n)$。 [24], 定义了共享出行的统一路线规划问题，并证明不存在能够以常数竞争比解决该问题的多项式时间算法；为近似求解该问题，研究提出了一种基于动态规划插入的两阶段解法。然而，这些研究都没有考虑动态交通情况下道路网络的时变出行成本特征。回到示例 1，现有插入算子可以通过直接累加静态道路网络中的四个静态边权重（o_2，o_3、o_3，d_2、d_2，d_3 和 d_3，d_1），计算新路线 o_1，o_2，o_3，d_2，d_3，d_1 的行程时间。然而，在时变道路网络中，这条新路线的行程时间取决于工作人员接载 r_3 的时间，也就是说，取决于 $arr(o_3)$。因此，现有插入算子中计算新路线行程时间的方法不再适用，现有插入算子无法高效处理时变道路网络中的路径规划相关问题。

6 结论

本文首先研究了时变道路网络上的实时拼车服务中广泛采用的一种插入算子。我们将现有插入算子扩展到时变场景，作为基线方法，该方法需要执行大量最短旅行时间查询，时间复杂度为 $O(n^3)$。我们进一步复合工作人员可行路线上的时变边权函数，得到复合旅行时间函数，使时间复杂度降至 $O(n^2)$。为维护属于工作人员路线的任意顶点对之间的复合旅行时间函数，空间开销也为 $O(n^2)$。基于时变道路网络的 FIFO性质，我们提出了一种高效的时变插入算子，其时间和空间复杂度均为线性。我们证明，给定路线中用于插入新请求目的地的位置，可以在 $O(1)$ 时间内找到插入其起点的最优位置。在真实数据集上的大量实验验证了该时变插入算子的效率和可扩展性。尤其是在两个真实世界数据集的不同设置下，该时变插入算子的速度最高可达基线方法的 25.79 倍。

参考文献

[1] 2021. 滴滴出行。http://www.didichuxing.com/ [2] 2021. GAIA。https://outreach.didichuxing.com/research/opendata/ [3] 2021. OpenStreetMap. http://www.openstreetmap.com/ [4] 2021. Uber. https://www.uber.com/ [5] 2022. BlaBlaCar. https://www.blablacar.co.uk/ [6] Di Chen、Ye Yuan、Wenjin Du、Yurong Cheng 和 Guoren Wang。2021。时变道路网络上的在线路线规划。载于 2021年IEEE第37届数据工程国际会议（ICDE），325–335。[7] Lu Chen、Qilu Zhong、Xiaokui Xiao、Yunjun Gao、Pengfei Jin 和 Christian S. Jensen。2018。价格与时间感知的动态拼车。载于 ICDE，1061–1072。[8] Peng Cheng、Hao Xin 和 Lei Chen。2017。道路网络上的效用感知拼车。载于 SIGMOD，1197–1210。[9] Camila F. Costa 和 Mario A. Nascimento。2021。考虑时变位置的最后一英里配送。载于 SIGSPATIAL，121–132。[10] Tianlun Dai、Bohan Li、Ziqiang Yu、Xiangrong Tong、Meng Chen 和 Gang Chen。2021。PARP：一种由并行交通状况驱动的动态道路网络路径规划模型。ACM 智能系统技术汇刊 12，6（2021），73:1–73:24。[11] Bowen Du、Yongxin Tong、Zimu Zhou、Qian Tao 和 Wenjun Zhou。2018。电动汽车共享的需求感知充电桩规划。载于 KDD，1330–1338。[12] Dawei Gao、Yongxin Tong、Jieying She、Tianshu Song、Lei Chen 和 Ke Xu。2016。空间众包中的 Top-k 团队推荐。载于 WAIM，191–204。[13] Wesam Mohamed Herbawi 和 Michael Weber。2012。一种用于解决带时间窗的动态拼车问题的遗传与插入启发式算法。载于 GECCO，385–392。[14] Yan Huang、Favyen Bastani、Ruoming Jin 和 Xiaoyang Sean Wang。2014。道路网络上提供服务保障的大规模实时拼车。VLDB基金会会刊 7，14（2014），2017–2028。[15] Zhidan Liu、Zengyang Gong、Jiangzhou Li 和 Kaishun Wu。2020。移动性感知的动态出租车拼车。载于 2020 IEEE第36届数据工程国际会议（ICDE），961–972。[16] Shuo Ma、Yu Zheng 和 Ouri Wolfson。2013。T-share：大规模动态出租车拼车服务。载于 ICDE，410–421。[17] Shuo Ma、Yu Zheng 和 Ouri Wolfson。2015。实时城市规模出租车拼车。IEEE知识与数据工程汇刊 27，7（2015），1782–1795。[18] Douglas O. Santos 和 Eduardo C. Xavier。2013。动态出租车与拼车：优化问题的框架与启发式方法。载于 IJCAI，2885–2891。[19] Jieying She、Yongxin Tong、Lei Chen 和 Tianshu Song。2017。反馈感知的社交活动参与者安排。载于 SIGMOD，851–865。[20] Qian Tao、Yuxiang Zeng、Zimu Zhou、Yongxin Tong、Lei Chen 和 Ke Xu。2018。实时空间众包中的多工作人员感知任务规划。载于 DASFAA，301–317。[21] Yongxin Tong、Ye Yuan、Yurong Cheng、Lei Chen 和 Guoren Wang。2017。时空众包数据管理技术综述。《软件学报》28，1（2017），35–58。[22] Yongxin Tong、Yuxiang Zeng、Bolin Ding、Libin Wang 和 Lei Chen。2021。空间众包中的双边在线微任务分配。IEEE知识与数据工程汇刊 33，5（2021），2295–2309。[23] Yongxin Tong、Yuxiang Zeng、Zimu Zhou、Lei Chen 和 Ke Xu。2022。共享出行的统一路径规划：一种基于插入的框架。ACM 数据库系统汇刊 47，1（2022），2:1–2:48。[24] Yongxin Tong、Yuxiang Zeng、Zimu Zhou、Lei Chen、Jieping Ye 和 Ke Xu。2018。共享出行路径规划的统一方法。VLDB基金会会刊 11，11（2018），1633–1646。[25] Yongxin Tong、Zimu Zhou、Yuxiang Zeng、Lei Chen 和 Cyrus Shahabi。2020。空间众包：综述。VLDB期刊 29，1（2020），217–250。[26] Yong Wang、Guoliang Li 和 Nan Tang。2019。查询时变道路网络中的最短路径。VLDB基金会会刊 12，11（2019），1249–1261。[27] Y. Wang、Y. Yuan、H. Wang、X. Zhou、C. Mu 和 G. Wang。2021。大型多模态时变网络上的受约束路径规划。载于 ICDE，313–324。[28] Yi Xu、Yongxin Tong、Yexuan Shi、Qian Tao、Ke Xu 和 Wei Li。2019。动态拼车服务中的高效插入算子。载于 ICDE，1022–1033。[29] Yi Xu、Yongxin Tong、Yexuan Shi、Qian Tao、Ke Xu 和 Wei Li。2022。动态拼车服务中的高效插入算子。IEEE知识与数据工程汇刊 34，8（2022），3583–3596。[30] Yuxiang Zeng、Yongxin Tong、Yuguang Song 和 Lei Chen。2020。越简单越好：共享路线规划查询的索引方法。VLDB基金会会刊 13，13（2020），3517–3530。[31] Boming Zhao、Pan Xu、Yexuan Shi、Yongxin Tong、Zimu Zhou 和 Yuxiang Zeng。2019。按需出租车派遣中的偏好感知任务分配：一种在线稳定匹配方法。载于 AAAI，2245–2252。[32] Libin Zheng、Lei Chen 和 Jieping Ye。2018。价格感知拼车中的订单派遣。VLDB基金会会刊 11，8（2018），853–865。[33] Libin Zheng、Peng Cheng 和 Lei Chen。2019。拼车中的基于拍卖的订单派遣与定价。载于 2019 IEEE第35届数据工程国际会议（ICDE），1034–1045。