适用于多个上车和下车地点的拼车快速多对多路径规划

Moritz Laupichler 卡尔斯鲁厄理工学院理论信息学研究所、算法工程，Kaiserstraße 12，76131 卡尔斯鲁厄，德国

Peter Sanders 卡尔斯鲁厄理工学院理论信息学研究所、算法工程，Kaiserstraße 12，76131 卡尔斯鲁厄，德国

arXiv:2305.05417v1 [cs.DS] 9 May 2023

摘要

我们介绍了 KaRRi，这是一种改进的算法，用于调度共享车辆车队，适用于 UberXShare 和 Lyft Shared 等服务。我们加速了基础在线算法，该算法会寻找将新客户插入一组现有路线的所有可能方式；同时，我们对目标函数进行了泛化，并高效支持大量可能的上车和下车地点。这为具有更高车辆载客率的拼车系统奠定了算法基础，从而能够在保持相当服务质量的同时，大幅降低成本和生态影响。我们发现，与此前最先进的方法相比，我们的算法计算车辆与乘客之间分配方案的速度提高了数倍。此外，我们观察到，为车辆和乘客设置会合点，可以将车队的运营成本降低最多 15%，同时缩短乘客的等待时间和行程时间。

2012年 ACM 学科分类 应用计算 → 《Transportation》；计算理论 →在线算法；计算理论 → 向量/流算法；信息系统→ 地理信息系统

关键词和短语 算法工程，路径规划，拼车，多模式

1 引言

当前的交通运输系统主要建立在个人交通（通常使用沉重、污染严重且消耗大量能源和空间的汽车）与公共交通的组合之上，而公共交通往往速度缓慢、使用不便且发展不足。能够智能控制大型出租车式车辆车队的拼车系统，有望提供一种比公共交通更便利、比个人使用汽车更经济环保的替代方案 [2, 15, 18]。如果这些车辆采用电力驱动并具备自动驾驶能力，这一前景尤其值得期待。然而，当前的此类系统并未兑现这些承诺，因为车辆的实际可用容量相当小，甚至可能导致行驶车辆的总公里数增加增加 而不是减少 [14]。对于容量更大的拼车车辆而言，一个主要问题是接送客户会给车辆中的其他乘客造成较大的延误。

本文为更好地将拼车车队整合到多模式交通系统中奠定算法基础。我们关注的问题是，如何利用本地交通方式（例如步行、自行车或滑板车）前往一个能减少车辆延误、可与其他客户共享，且可能减轻乘客隐私顾虑的取件或下车地点（*PD-* 位置） [16, 10]。（接下来相关的一步将分析如何利用火车或快速公交车等公共交通方式更快、更经济／环保地跨越长距离，从而有效运用个人交通、拼车和公共交通这一层级体系。）

我们的出发点是 Buchhold等人 [4] 提出的 LOUD系统，它包含一个面向大型拼车车队的在线调度系统。该系统使用基于桶式收缩层次（BCH）的 一对多路径规划 [13, 9] ，以高效地找出将新客户插入车辆当前时刻表的最佳方案。这是实时管理大型车队，以及在交通研究中计算此类系统逼真模拟结果的关键步骤。

我们介绍 KaRRi（Karlsruhe Rapid Ridesharing）算法，该算法扩展了 LOUD，使乘客能够在其出发地和目的地附近的任意位置上车和下车，而不局限于固定地点。该算法会计算乘客与车辆之间的最优分配，其中也包括上车地点和下车地点。我们通过纳入乘客等待时间、行程时间，以及乘客前往上车地点和从下车地点离开所需的个人交通开销，将 LOUD 的目标函数调整为适用于这一新场景。

要找到最适合某个请求的车辆，还要确定车辆、上车地点和下车地点的最优组合，这会使可能的分配数量大幅增加。为确定最佳分配，我们需要解决车辆位置与所有 可能的接送地点之间的多个多对多路径规划问题。我们使用 BCH 查询来应对这一问题，并针对通用的基于桶的查询以及源点或目标点局部集中的特定情况，提出了新的加速技术。我们发现，这些技术也适用于加快单个上车点和下车点情况下的路径规划。

我们的实验评估使用真实数据集来评估这些措施的效率。我们发现，在只有一个取件点和下车点的情况下，我们的实现速度比 LOUD 快数倍。对于多个接送地点，我们的路由技术比 LOUD 技术的朴素扩展快最多三个数量级。我们还初步发现，在不增加乘客等待时间或行程时间的情况下，允许设置多个接送地点可以将出租车车队的运营成本降低最多 15%。未来工作可能会与应用领域专家合作，进一步深入研究其对运输系统可能产生的影响。

论文概述。 在第2节对问题进行更详细的说明之后，我们将在第3节介绍基本符号和技术。第4节将在存在多个接送地点的情况下，为我们的成本模型定义形式化基础。第5节概述 KaRRi 算法，第6至第8节则分别讨论各项具体挑战。在第9节中，我们将通过实验评估我们的方法。

2 问题陈述

本节将介绍并正式阐述我们的方法所考虑的动态共享乘车问题。

道路网络。 我们将道路网络视为一个图 $G = (V_G, E_G)$，其中边表示道路路段，顶点表示交叉口。每条边 $e = (v, w) \in E_G$ 都有一个行驶时间 $\ell(e) = \ell(v, w)$。我们将从顶点 v 到顶点 w 的最短路径距离（即行驶时间）记为 $\delta(v, w)$。我们的算法使用两个相互独立的道路网络 G_{veh} 和G_{psg} 及其相关的 ℓ_{veh}、δ_{veh}、ℓ_{psg}和 δ_{psg}，分别表示同一道路网络中车辆和行人可通行的部分。请注意，我们仅考虑步行，但G_{psg} 也可以表示其他交通方式，例如骑行。如果某个位置 v 在两个网络中均可通行，即 v，则乘客可以在该位置上下车 $v \in V_{veh} \cap V_{psg}$。

车辆、停靠点。我们的算法可以使用一支由 F 辆车辆组成的车队。每辆车辆 $\nu = (l_i, \text{cap}, t^{min}_{serv}, t^{max}_{serv})$都有一个初始位置 l_i、座位容量 cap 和服务时间区间$[t^{min}_{serv}, t^{max}_{serv}]$。车辆的当前路线 $R(\nu) = s_0(\nu), \ldots, s_{k(\nu)}(\nu)$是为该车辆安排的一系列停靠点。车辆的当前位置 $l_c(\nu)$始终位于其上一个（或当前）停靠点 $s_0(\nu)$与下一个停靠点 $s_1(\nu)$之间。因此， $k(\nu) = |R(\nu)| - 1$ 表示该车辆尚需访问的停靠点数量。每个停靠点 s 都映射到图中的一个顶点$l(s) \in V$。为简化记法，我们可以用 s_i 代替 $s_i(\nu)$，并仅用 s_i 代替 $l(s_i)$。在每个停靠点，车辆接载和/或下车一名或多名乘客，并至少停留 t^{min}_{stop} 的时间，该时间是一个模型参数。我们用 $o(s_i)$ 表示车辆在停靠点 s_i 和 s_{i+1} 之间的载客量。随着为新分配的乘客添加停靠点，我们会更新每辆车辆的路线。对于每个停靠点 $s \in R(\nu)$，我们根据当前时刻表维护最早可能的到达时间 $t^{min}_{arr}(s)$和出发时间 $t^{min}_{dep}(s)$。

请求。 在我们的场景中，调度器接收乘车请求，并立即将其分配给车辆。请求 $r = (\text{出发地}, \text{目的地}, t_{req})$包含一个起点位置 出发地 $\in V$、一个目的地位置 目的地 $\in V$，以及请求发出的时间 t_{req}。我们不允许预订，即请求时间也是最早可能的出发时间。

上车地点（位置）、下车地点（位置）。 可能的上车地点（简称上车点）是一个位置 $v \in V_{psg} \cap V_{veh}$，车辆可在时间半径 ρ内于 G_{psg} 从出发地(r)到达该位置。类似地，可能的下车地点（下车点）是一个位置 $v \in V_{psg} \cap V_{veh}$，车辆可在 ρ内于 G_{psg} 从该位置到达目的地(r)。 r 对应的接客点和下客点集合，以及半径 ρ，分别记为 $P_\rho(r)$ 和 $D_\rho(r)$。令 $N_p(r) = |P_\rho(r)|$，且 $N_d(r) = |D_\rho(r)|$。我们将接客点和下客点统称为接送地点。我们将一对上车地点和下车地点称为一个*PD* 对。半径 ρ 是一个模型参数。

插入。 调度器的目标是为每个请求 r 找到将其接送插入任意车辆路线的方法，使根据成本函数计算的该插入成本最小。我们将插入形式化为一个元组 (r, p, d, ν, i, j)，表示车辆 ν 在停靠点 $s_i(\nu)$ 之后立即于上车地点 $p \in P_\rho(r)$ 接送请求 r，并在停靠点 $s_j(\nu)$ 之后立即于下车地点 $d \in D_\rho(r)$ 让其下车，其中包含 $0 \leq i \leq j \leq k(\nu)$。

2.1 成本函数与约束

插入 $\iota = (r,\ p,\ d,\ \nu,\ i,\ j)$ 的成本考虑了新增的车辆运营时间 $t_{detour}(\iota)$（对应 ν）、 r 的行程时间 $t_{trip}(\iota)$、现有乘客 ν 的行程时间增加总和 $t^{+}_{trip}(\iota)$，以及步行时间 $t_{walk}(\iota)$（我们将这些术语的确切定义推迟到第4节）。

我们考虑 [4] 中提出的若干适用于符合条件的插入的约束。插入后，必须满足以下条件：首先，插入操作不得使 ν 的载客量超限。其次，车辆必须仍能在服务时间结束前到达其最后一站。第三，已分配给 ν 的每位乘客仍须在最长等待时间 t^{max}_{wait}内，在其取件停靠点上车。第四，已分配给 ν 的每位乘客 $\hat{r}$ 仍须在最长行程时间 $t^{max}_{trip}(\hat{r}) = \alpha \cdot \delta_{veh}$(原点($\hat{r}$), 目的地($\hat{r}$)) $+ \beta$内到达其目的地。 t^{max}_{wait}、 α 和 β 的值为模型参数。

对于已分配给 ν 的请求，这四项约束均为硬约束。如果 ι 违反硬约束，我们将成本设为 ∞。对于待插入的请求 r，我们将等待时间和行程时间约束视为软约束，也就是说，违反这些约束会产生成本惩罚。假设乘客在时间 t_{dep}于 p 上车。我们将成本惩罚定义为

$$c^{vio}_{wait}(\iota) = \gamma_{wait} \cdot \max\{t_{dep} - t_{req}(r) - t^{max}_{wait}, 0\}$$
$$c^{vio}_{trip}(\iota) = \gamma_{trip} \cdot \max\{t_{trip}(\iota) - t^{max}_{trip}(r), 0\}$$

其中模型参数 γ_{wait} 和 γ_{trip} 用于调整惩罚项的严重程度。

对于总插入成本，我们采用车辆绕行时间、乘客行程时间、步行时间以及软约束违反惩罚项的线性组合：

$$c(\iota) = t_{detour}(\iota) + \tau \cdot (t_{trip}(\iota) + t^{+}_{trip}(\iota)) + \omega \cdot t_{walk}(\iota) + c^{vio}_{wait}(\iota) + c^{vio}_{trip}(\iota)$$

请注意，我们的成本函数基于 LOUD 算法 [4] 中使用的成本函数。然而，原始成本函数未考虑乘客行程时间或步行时间。我们通过模型参数 τ 和 ω 对这些时间的重要性进行加权。当 $\tau = \omega = 0$ 时，我们的成本函数与 LOUD 的成本函数等价。

3 预备知识

本节介绍几种用于计算最短路径的算法，这些算法将在本文中使用。此外，我们还总结了用于动态拼车的 LOUD 算法 [4]，该算法构成了我们工作的基础。

3.1 最短路径算法

接下来，我们将介绍多种算法，用于计算道路网络中不同类型的最短路径查询。

迪杰斯特拉最短路径算法。 迪杰斯特拉最短路径算法 [8]计算加权图中从源点 $s \in V$到所有其他顶点的最短路径 $G = (V, E, \ell)$。

该算法会为每个 $v \in V$存储一个距离标签 $\delta(s, v)$。一个以 $v = \delta(s, v)$ 为键的可寻址优先队列（PQ） Q 包含活动顶点。初始时， $Q := \{s\}$， $\delta(s, s) := 0$，且 $\delta(s, v) := \infty$，其中 $v \neq s$。该算法通过从 Q 中提取距离标签最小的顶点并将其标记为已确定来推进。要确定 $u \in V$，需对其每条出边 $(u, v) \in E$ 进行松弛。对 $e = (u, v)$的松弛会尝试使用 $\delta(s, u) + \ell(e)$ 改进距离标签 $\delta(s, v)$。如果距离得到改进，就将 v 插入 Q。当 Q 变为空时，算法停止。

收缩层次。 收缩层次 *(CHs)* [9] 是一种用于道路网络中最短路径计算的加速技术，它利用了道路网络的层次结构。收缩层次在预处理阶段构建。之后，可以在收缩层次上使用受限的 Dijkstra 搜索计算最短路径查询。

为了构建收缩层次，道路网络 $G = (V, E)$ 中的所有顶点会根据其重要性或等级 [9]进行启发式排序。顶点按等级从低到高依次收缩。收缩 $v \in V$ 会暂时从图中移除 v 。为保留最短路径，如果 $(u, v, w) \in E^2$ 是 u 与 w之间唯一的最短路径，则会创建一条捷径边(u, w)。

令 E^+ 包含所有原始边 E 以及所有捷径边。图 $G^+ = (V, E^+)$ 构成收缩层次。捷径边 e 的长度 $\ell^+(e)$ 是被替换的原始边长度之和，而 δ^+ 是相应的距离函数。在查询阶段，我们将 E^+ 划分为上行边 $E^\uparrow = \{(u, v) \in E^+ \mid \text{等级}(u) < \text{等级}(v)\}$ 和下行边$E^\downarrow = \{(u, v) \in E^+ \mid \text{等级}(u) > \text{等级}(v)\}$。我们定义上行搜索图 $G^\uparrow := (V, E^\uparrow)$ 和下行搜索图 $G^\downarrow := (V, E^\downarrow)$。距离函数 $\delta^\uparrow$ 和 $\delta^\downarrow$ 分别表示受 $G^\uparrow$ 和 $G^\downarrow$约束的 δ^+。以顶点$v \in V$ 为根的上行收缩层次搜索空间 $G^\uparrow_v = (V^\uparrow_v, E^\uparrow_v)$ 包含所有可从 v 在 $E^\uparrow$ 中到达的顶点 $V^\uparrow_v \subseteq V$，以及相应的边 $E^\uparrow_v \subseteq E^\uparrow$。以顶点$v \in V$为根的反向下行收缩层次搜索空间 $G^\downarrow_v = (V^\downarrow_v, E^\downarrow_v)$ 包含所有能够在 $E^\downarrow$ 中到达 v 的顶点 $V^\downarrow_v \subseteq V$，以及相应的边$E^\downarrow_v \subseteq E^\downarrow$。

对于任意两个顶点 $s, t \in V$，可以证明，从 s 到 t 存在一条最短路径，该路径在收缩层次中是上升-下降路径，即仅由上升边后接仅由下降边组成 [9]。从源点 $s \in V$ 到目标点 $t \in V$ 的收缩层次查询 会从 s 在$G^\uparrow$ 中执行正向 Dijkstra 搜索，并从 t 在 $G^\downarrow$中执行反向 Dijkstra 搜索。搜索相遇时，它们会找到一条从 s 到 t的上升-下降路径，并最终找到一条最短路径。当任一 Dijkstra 搜索的半径超过此前找到的从 s 到 t的最佳距离后，查询即可停止。

桶式收缩层次搜索。 桶式收缩层次 *(BCH)* [13, 9]搜索用于查找道路网络 $G = (V, E)$ 中从源点 $s \in V$ 到一组目标点 $T \subseteq V$ 的所有最短路径距离。算法以 G 的收缩层次 G^+ 为基础。

其思路是在每个顶点 $v \in V$ 处构建一个（目标）桶 $B^\downarrow(v)$。对于每个目标点$t \in T$，在 $G^\downarrow$ 中运行一次反向搜索，并为每个已确定顶点$v \in V$向 $B^\downarrow(v)$中添加一条条目 $(t, \delta^\downarrow(v, t))$ 。然后，在 $G^\uparrow$ 中从 s 开始进行正向搜索，即可为每个已确定顶点 $v \in V^\downarrow_s$处的每个桶条目 $(t, \delta^\downarrow(v, t))$ 计算暂定最短路径距离 $\delta^\uparrow(s, v) + \delta^\downarrow(v, t)$。

BCH 搜索也可以类似地计算从一组源点到单个目标点的距离。在这种情况下，我们将每个 $v \in V$对应的源桶 $B^\uparrow(v)$称为源桶。

BCH 搜索的优点是，每个源点和每个目标点的搜索空间都只需遍历一次，可以用于计算桶条目或扫描桶条目。然而，存储桶条目比单独进行点对点 CH 查询需要更多内存。

捆绑式搜索。 针对多个源点的基于 Dijkstra 的最短路径算法可以使用捆绑式搜索 ，同时推进从 k 个源点出发的搜索。捆绑式搜索会为每个顶点维护 k 个暂定距离标签。搜索以每个 k 源点为起点。初始化时，第 j个源点处的第 j个距离标签设为 0，所有其他 $kn - k$ 距离标签均设为 ∞。与通常一样，通过松弛每条出边来确定顶点。当搜索松弛一条边 $(u, v) \in E$ 时，它会尝试更新 v处的所有 k 距离标签。

捆绑松弛的缓存效率可能高于 k 单独松弛，因为所有k 距离都存储在连续内存中。不过，如果并非所有 k 搜索都已到达 u ，那么对 $(u, v) \in E$ 的松弛可能会产生无效工作。因此，只有当所有 k 搜索松弛的边大致相同时，捆绑才有效。 k 的值是一个调优参数。

捆绑式搜索这一概念最初用于计算弧标记的 Dijkstra 搜索，当时称为集中式搜索 [11]。此后，捆绑式搜索已应用于多种基于 Dijkstra 的多对多最短路径算法 [1, 17,7, 5, 6]。最近，这一思路也扩展到了收缩层次（CH）中的点对点查询 [3]。

捆绑式搜索中的指令级并行性。 此外，还可以利用单指令多数据（SIMD）并行性来执行捆绑式搜索 [3]。现代 CPU 提供了专用的向量寄存器和指令，可同时存储和处理多个数据项。我们可以将边松弛过程中所需的计算向量化，使得 k计算通过一条向量指令同时完成。SIMD 指令可以大幅加快捆绑式搜索 [3]。

3.2 LOUD

我们的算法基于动态拼车调度算法$LOUD$ [4]。

给定一支车队及其路线，在线算法会将新到达的拼车请求分配给车辆。对于每个请求，都会找到一种可行的插入方式，将请求的出发地 o 和目的地 d 纳入车辆的路线，从而最大限度地减少车辆的绕行距离。

椭圆剪枝。 为计算可能插入方案的成本，算法需要计算现有车辆停靠点与 o 和 d之间的距离。LOUD 使用 BCH，为每个车辆停靠点设置桶条目，并从 o 和 d发起查询，以计算这些距离。

我们将这些 BCH 搜索称为椭圆 *BCH* 搜索，因为它们采用了一种针对这些桶的剪枝技术，称为椭圆剪枝：每次插入都必须满足我们在第 2.1 节中描述的相同软约束和硬约束。已分配给车辆 $\nu \in F$ 的乘客的等待时间和行程时间硬约束定义了余量 $\lambda(s_i,\ s_{i+1})$，即每对连续停靠点之间 $(s_i,\ s_{i+1}) \in R(\nu)$ 所允许的最大绕行距离。任何超过 $\lambda(s_i,\ s_{i+1})$ 的绕行都会违反某项硬约束，因此不可行。余量$\lambda(s_i,\ s_{i+1})$定义了一个绕行椭圆，其中包含所有可以在 s_i 与 s_{i+1} 之间进行取件或下车而不违反硬约束的顶点。因此，只需在椭圆内的顶点处为 s_i 和 s_{i+1} 生成桶条目。椭圆剪枝大幅减少了 BCH 搜索需要扫描的桶条目数量，并限制了可用于插入的候选车辆数量 [4]。

最后一站距离。LOUD 还允许在车辆路线的最后一站之后插入出发地和/或目的地。在这种情况下，椭圆剪枝并不适用，因为车辆在最后一站之后的宽限范围没有上限。相反，LOUD 会在以 o 或 d 为根的道路网络中执行反向 Dijkstra 查询，以计算从最后一站到 o 或 d 的距离。这些 Dijkstra 查询，尤其是计算从最后一站到请求目的地的距离，占据了 LOUD 总运行时间的很大一部分（至少 60%，最高超过 90%）。

4 多个上车和下车地点的概念变化

我们发现，为上车和下车地点引入乘客移动时，必须仔细考虑其对车辆绕行和乘客行程的影响。下文将详细描述这些影响，并据此为我们的成本函数奠定形式化基础。

请注意，为简化起见，本节忽略了两种特殊情况：第一，我们假设插入 $\iota = (r,\ p,\ d,\ \nu,\ i,\ j)$ 时，接送均插入在 ν 的下一个停靠点之后，即 $i \geq 1$。第二，我们不考虑 p 或 d 与现有停靠点重合的可能性，即假设 $p \neq l(s_i)$ 和 $d \neq l(s_j)$。我们忽略这些情况，因为它们

会导致定义过于冗长。不过，若已知车辆当前位置$l_c(\nu)$以及每个停靠点 $s \in R(\nu)$ 的位置 $l(s)$，则可以很直接地将这些被忽略的情况纳入以下定义中。

4.1 步行时间与步行至目的地

常规插入 $\iota = (r, p, d, \nu, i, j)$ 的步行时间仅为 $t_{walk}(\iota) := \delta_{psg}(orig\ p) + \delta_{psg}(d, dest)$。

我们允许每位乘客 r 从出发地步行前往目的地，全程不乘坐任何车辆。这需要一种伪插入方式 ι_{psg}，即乘客完全不与任何车辆匹配。此时，ι_{psg} 的成本仅取决于步行距离$t_{walk}(\iota_{\text{psg}}) = t_{trip}(\iota_{\text{psg}}) = \delta_{psg}$(出发地，目的地)。伪插入不会影响车辆运营时间，也不会影响其他乘客的行程时间。由于乘客无需等待车辆，我们忽略等待时间软约束。实际上，总成本为 $c(\iota_{\text{psg}}) = \tau \cdot t_{trip}(\iota_{\text{psg}}) + \omega \cdot t_{walk}(\iota_{\text{psg}}) + c^{vio}_{trip}(\iota_{\text{psg}})$。对于任意距离 δ_{psg}(出发地，目的地)，我们都明确允许伪插入，也就是说，该距离不必在 出发地 或 目的地周围的半径 ρ 范围内找到。相反，我们使用乘客图中的收缩层次查询来查找 δ_{psg}(出发地，目的地)。成本 $c(\iota_{\text{psg}})$ 可作为任何插入成本的初始上界。

4.2 等待乘客的车辆

在传统的动态拼车模型中，请求 $r =$(出发地，目的地 t_{req}) 总是等待车辆在起点位置 出发地接载。请求在时间 t_{req}发出，而与该请求 ν 匹配的车辆最早可在 t_{req}开始前往上车地点。如果上车地点是出发地，这意味着车辆总是比乘客晚到达上车地点，也就是说，只有乘客会等待车辆，而不会反过来。因此，车辆在任何停靠点停留的时间总是恰好为 t^{min}_{stop}（除非车辆处于空闲状态，即当前没有其他停靠点需要前往）。所以，对于每个停靠点 s，都可以根据计划出发时间推断出计划到达时间，即 $t^{min}_{arr}(s) = t^{min}_{dep}(s) - t^{min}_{stop}$。在传统的动态拼车模型中，只需在完整的车辆调度计划 [4]中存储每个停靠点的出发时间即可。随后，两个相邻停靠点 s_i 和 s_{i+1} 之间的最短路径距离为$\delta_{veh}(s_i, s_{i+1}) = t^{min}_{dep}(s_{i+1}) - t^{min}_{stop} - t^{min}_{dep}(s_i)$。

现在，考虑一次插入 $\iota = (r, p, d, \nu, i, j)$，其中 $p \neq orig$。此时，车辆和乘客都必须前往 p，分别从 $t^{min}_{dep}(s_i)$ 和 t_{req} 出发。因此，当且仅当 $t^{min}_{dep}(s_i) + \delta_{veh}(s_i, p) < t_{req} + \delta_{psg}(orig\ p)$ 时，车辆才能比乘客更早到达 p。在这种情况下，车辆需要在 p 等待乘客。于是，在 p 的插入停靠点 s 可能比 t^{min}_{stop} 耗时更长。因此，我们将每个取件点实际最早可能的出发时间定义为车辆和乘客在该取件点各自最早可能出发时间的最大值。车辆 ν 最早可在停靠 t^{min}_{stop} 后从 p 出发，而乘客一到达 p 就可以出发，由此得到

$$t_{dep}(\iota) = \max(t^{min}_{dep}(s_i) + \delta_{veh}(s_i, p) + t^{min}_{stop}, t_{req} + \delta_{psg}(orig, p)).$$

稍后，我们会在定义车辆绕行、乘客等待时间和成本函数所需的乘客行程时间时使用 $t_{dep}(\iota)$（见第 2.1 节）。具体而言，车辆或乘客的等待时间会计入车辆运营时间和乘客行程时间。

而在传统模型中，我们可以根据停靠点 s 的 $t^{min}_{dep}(s)$ 推断出 $t^{min}_{arr}(s)$；现在则必须显式存储 $t^{min}_{arr}(s)$ 和 $t^{min}_{dep}(s)$。随后即可推导出一对相邻停靠点（s_i、s_{i+1}）之间的距离，即 $\delta_{veh}(s_i、s_{i+1}) = t^{min}_{arr}(s_{i+1}) - t^{min}_{dep}(s_i)$。

我们用 $t^{veh}_{wait}(s) = t^{min}_{dep}(s) - t^{min}_{arr}(s) - t^{min}_{stop}$ 表示车辆在停靠点 s 的计划等待时间。

4.3 增加的车辆运营时间和乘客行程时间

下面，我们定义插入 $\iota = (r, p, d, \nu, i, j)$ 时增加的车辆运营时间 $t_{detour}(\iota)$、新乘客的行程时间 $t_{trip}(\iota)$，以及现有乘客增加的行程时间总和 $t^{+}_{trip}(\iota)$。第4.2节中所述的车辆等待时间会影响所有这些时间。

为了说明这种影响，我们首先关注由插入 ι 引起的车辆运营时间增加。从概念上讲，车辆运营时间的增加是指插入后车辆到达最后一个计划停靠点的到达时间与插入前到达最后一个计划停靠点的到达时间之间的差值。

初始绕行。 我们首先说明在没有车辆等待时间的情况下，即对所有 $s \in R(\nu)$，$t^{veh}_{wait}(s) = 0$，车辆运营时间如何增加。在这种情况下，在 p 取件并在 d 下车所增加的车辆运营时间，分别等于车辆 ν 在停靠点 s_i 和停靠点 s_j 之后的路线中必须绕行的时间。我们将这些绕行分别称为初始接客绕行和初始下车绕行。

▶ **定义 1.** 插入 $\iota = (r, p, d, \nu, i, j)$ 所产生的初始接客绕行（下车绕行），是指车辆 ν 先驶往停靠点 s_i (s_j) 之后的 p (d)，而不是直接驶往 s_{i+1} (s_{j+1})，由此产生的绕行。

形式化地，初始接客绕行定义如下：

$$\Delta^{init}_{p}(\iota) := \begin{cases} t_{dep}(\iota) - t^{min}_{dep}(s_i) & \text{if } i = j \\ t_{dep}(\iota) - t^{min}_{dep}(s_i) + \delta_{veh}(p, s_{i+1}) - \delta_{veh}(s_i, s_{i+1}) & \text{if } i \neq j \end{cases}$$

初始下车绕行定义如下：

$$\Delta^{init}_{d}(\iota) := \begin{cases} \delta_{veh}(p, d) + t^{min}_{stop} & \text{if } i = j \text{ and } j = k(\nu) \\ \delta_{veh}(s_j, d) + t^{min}_{stop} & \text{if } i \neq j \text{ and } j = k(\nu) \\ \delta_{veh}(p, d) + t^{min}_{stop} + \delta_{veh}(d, s_{j+1}) - \delta_{veh}(s_j, s_{j+1}) & \text{if } i = j \text{ and } j \neq k(\nu) \\ \delta_{veh}(s_j, d) + t^{min}_{stop} + \delta_{veh}(d, s_{j+1}) - \delta_{veh}(s_j, s_{j+1}) & \text{if } i \neq j \text{ and } j \neq k(\nu) \end{cases}$$

请注意，如果 $i = j$，车辆必须先从停靠点 s_i 驶至 p，再驶至 d，最后驶至 s_{i+1}，从而形成组合绕行。在这种情况下，现有路线 $R(\nu)$ 中只有 s_i 与 s_{i+1} 之间的路段会被组合绕行替代。初始接客和下车绕行的定义考虑了这一点：如果 $i = j$，则仅在下车绕行中减去一次 s_i 与 s_{i+1} 之间的距离，而不在接客绕行中减去。

剩余绕行、增加的车辆运营时间。 如果没有车辆等待时间，则车辆在接客点 p 的出发时间 $t_{dep}(\iota)$ 是车辆 ν 到达 p 的到达时间加上最短停靠时间t^{min}_{stop}，因此 $t_{dep}(\iota) = t^{min}_{dep}(s_i) + \delta(s_i, p) + t^{min}_{stop}$（见第4.2节）。车辆等待时间对绕行的第一个影响是，若乘客到达接客点的时间晚于车辆，则 $t_{dep}(\iota)$ 可能取决于乘客。如第4.2节所述，在乘客移动的情况下，我们有 $t_{dep}(\iota) = \max(t^{min}_{dep}(s_i) + \delta(s_i, p) + t^{min}_{stop}, t_{req}(r) + \delta_{psg}(\text{原始 } p))$。因此，在有车辆等待时间的情形下，初始接客绕行 $\Delta^{init}_{p}(\iota)$ 可能大于无车辆等待时间时的绕行。

此外，$R(\nu)$中停靠点处现有的车辆等待时间也可能计入新增的车辆运营时间。假设 $j + 1 < a < k(\nu)$、$t^{veh}_{wait}(s_a) > 0$ 以及 $t^{veh}_{wait}(s_b) = 0$ 对于$b \neq a$。令 $t^{min}_{arr'}(s, \iota)$和 $t^{min}_{dep'}(s, \iota)$ 分别表示计划到达时间和出发时间

在执行插入 ι后，于停靠点 $s \in R(\nu)$。插入后，车辆 ν 到达停靠点 s_{j+1} 时将延迟 $\Delta_{j+1}^{res}(\iota) = \Delta_p^{init}(\iota) + \Delta_d^{init}(\iota)$，因此，与 $= t_{arr}^{min}(s_{j+1}) + \Delta_{j+1}^{res}(\iota)$相比， $t_{arr}^{min\prime}(s_{j+1})$。这一到达时间延迟会沿路线向前传播，直到停靠点 s_a，因此对于到达 s_a 时的延迟，我们有 $\Delta_a^{res}(\iota) = \Delta_{j+1}^{res}(\iota)$。然而，在插入之前，车辆必须在停靠点 s_a等待 $t_{wait}^{veh}(s_a)$，以接载计划在此上车的乘客。由于车辆现在较晚到达 s_a，等待时间可能会缩短。如果 s_a 处的延迟大于 s_a处之前的等待时间，车辆甚至可能在 s_a 处晚于乘客到达。

实际上，车辆现在利用原本在 s_a 等待乘客所花费的时间，来完成接送 ι 所需的部分绕行。因此，插入操作的取件或下车点之后各停靠点的车辆等待时间，可以作为增加的车辆运营时间的缓冲。在我们的示例中，我们有

$$t_{arr}^{min\prime}(s_a, \iota) = t_{arr}^{min}(s_a) + \Delta_a^{res}(\iota) \text{ but}$$
$$t_{dep}^{min\prime}(s_a, \iota) = t_{dep}^{min}(s_a) + \Delta_{a+1}^{res}(\iota) \text{ where } \Delta_{a+1}^{res}(\iota) = \max(\Delta_a^{res}(\iota) - t_{wait}^{veh}(s_a), 0)$$

请注意，车辆在 s_a 的等待时间会缩短 s_a 的出发时间延迟，也会缩短其后所有停靠点的到达时间。形式化地，对于每个满足 $a < b$ 的停靠点 s_b，我们得到 $\Delta_b^{res}(\iota) = \Delta_{a+1}^{res}(\iota)$，因此 $t_{arr}^{min\prime}(s_b, \iota) = t_{arr}^{min}(s_b) + \Delta_{a+1}^{res}(\iota)$。

这引出了剩余绕行的定义，它描述插入 ι 对路线 $R(\nu)$上任一现有停靠点的到达时间造成的实际延误。它考虑到车辆可能在任一停靠点等待，从而减少所有后续停靠点的延误。

▶**定义 2. </style}插入 (**r, p, d, ν, i, j) 至停靠点$s_a \in R(\nu)$ 后，剩余绕行 $\Delta_a^{res}(\iota)$ 描述了车辆 ν 到达停靠点 s_a 的时间会延后多少。形式化地，我们将其定义为

$$\Delta_a^{res}(\iota) := \begin{cases} 0 & \text{if } a \leq i \\ \Delta_p^{init}(\iota) & \text{if } i+1 = a \leq j \\ \max(\Delta_j^{res}(\iota) - t_{wait}^{veh}(s_j), 0) + \Delta_d^{init}(\iota) & \text{if } a = j+1 \text{ and } i \neq j \\ \Delta_p^{init}(\iota) + \Delta_d^{init}(\iota) & \text{if } a = j+1 \text{ and } i = j \\ \max(\Delta_{a-1}^{res}(\iota) - t_{wait}^{veh}(s_{a-1}), 0) & \text{otherwise} \end{cases}$$

剩余绕行使我们能够轻松定义在每个停靠点 $s_a \in R(\nu)$插入 ι 后的新到达时间 $t_{arr}^{min\prime}$ 和出发时间$t_{dep}^{min\prime}$，如下所示

$$t_{arr}^{min\prime}(s_a, \iota) = t_{arr}^{min}(s_a) + \Delta_a^{res}(\iota) \text{ and}$$
$$t_{dep}^{min\prime}(s_a, \iota) = \max(t_{arr}^{min\prime}(s_a) + t_{stop}^{min}, t_{dep}^{min}(s_a)).$$

然后，新增的车辆运营时间可定义为:

▶**定义 3.** 由插入 $\iota = (r, p, d, \nu, i, j)$ 导致的车辆运营时间增加量 $t_{detour}(\iota)$ 定义为

$$t_{detour}(\iota) := \begin{cases} \Delta_p^{init}(\iota) + \Delta_d^{init}(\iota) & \text{if } i = j = k(\nu) \\ t_{arr}^{min\prime}(s_{k(\nu)}, \iota) - t_{arr}^{min}(s_{k(\nu)}) + \Delta_d^{init}(\iota) & \text{if } i < j = k(\nu) \\ t_{arr}^{min\prime}(s_{k(\nu)}, \iota) - t_{arr}^{min}(s_{k(\nu)}) & \text{otherwise} \end{cases}$$

行程时间、现有乘客增加的行程时间。 先前的定义还使我们能够定义与请求 r 相关联的乘客的行程时间 $t_{trip}(\iota)$。对于 r，行程时间就是请求时间 $t_{req}(r)$ 与乘客按时刻表到达其目的地 目的地(r) 之间的时间。从时间顺序上看，行程始于乘客前往上车点 p，并可能等待车辆到达 p。接下来，乘客乘坐车辆从 p 前往 d。最后，乘客从下车点 d 前往其目的地。

▶ **定义 4.** 插入 $\iota = (r, p, d, \nu, i, j)$ 的行程时间 $t_{trip}(\iota)$ 定义为

$$t_{trip}(\iota) := (t_{dep}(\iota) - t_{req}(r)) + t_{ride}(\iota) + \delta_{psg}(d, dest(r))$$

其中

$$t_{ride}(\iota) := \begin{cases} \delta_{veh}(p, d) & if\ i = j \\ \delta_{veh}(p, s_{i+1}) + (t^{min\prime}_{dep}(s_j, \iota) - t^{min\prime}_{arr}(s_{i+1}, \iota)) + \delta_{veh}(s_j, d) & if\ i \neq j \end{cases}$$

插入的总成本中的最后一项是为已分配给车辆 ν 的乘客增加的行程时间。考虑一个先前已分配给车辆 ν 的请求 r'。假设 r' 由 ν 在停靠点 $s_{i'}$ 上车，并在停靠点 $s_{j'}$ 下车。根据剩余绕行的定义，执行插入 ι 会使车辆到达 $s_{j'}$ 的时间延迟 $\Delta^{res}_{j'}(\iota)$。因此，r' 的行程时间也会增加相同的时长。

▶**定义 5.** 设 $N_d(s)$ 为车辆 $\nu \in F$ 当前时刻表中安排在停靠点 $s \in R(\nu)$ 进行下客的数量。插入 $\iota = (r, p, d, \nu, i, j)$ 所导致的现有乘客增加的行程时间之和$t^{+}_{trip}(\iota)$ 定义为

$$t^{+}_{trip}(\iota) := \sum_{a=i+1}^{k(\nu)} N_d(s_a) \cdot \Delta^{res}_a(\iota)$$

5 算法概述

我们介绍 KaRRi 算法（Karlsruhe Rapid Ridesharing），它利用快速多对多路径规划，高效处理包含多个接送地点的拼车请求。KaRRi 算法会动态接受请求，并根据成本函数和当前系统状态，为每个请求找到成本最优的插入方案。

对于请求 r，算法首先通过有界 Dijkstra 搜索，找出出发地和目的地步行半径ρ 范围内的可行接送地点。然后，算法按照图 1 所示的类型顺序评估所有插入方案。对于每个插入方案，KaRRi 会根据成本函数计算成本（见第 2.1 节）。成本最低的插入方案 ι^* 会不断更新，并最终返回。

由于我们考虑可能的接送地点集合，潜在插入的数量成为算法面临的主要挑战。具体而言，我们需要计算现有车辆停靠点与各接送地点之间的最短路径，以筛除不可行插入，并确定其余候选插入的成本。在以下各节中，我们将介绍所采用的捆绑和过滤方法，以限制每种插入类型所需的多对多最短路径查询的运行时间。

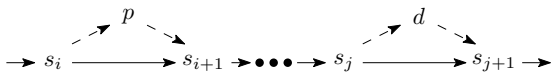
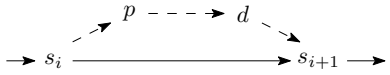
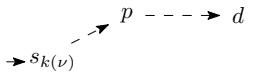
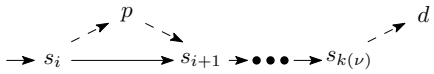
普通的	$0 < i < j < k(\nu)$	
普通的 成对（OP）	$0 < i = j < k(\nu)$	
取件前 上车（PBNS）	$0 = i \le j < k(\nu)$	
取件后 （PALS）	$0 \le i = j = k(\nu)$	
下车点之后 最后一站 (DALS)	$0 < i < j = k(\nu)$	

图 1 插入类型。根据一次插入中的取件和下车点插入位置 i 和 j，展示各类型的特征 $\iota = (r, p, d, \nu, i, j)$。图示描绘了 ν 当前的路线（实线箭头）及停靠点 $s \in R(\nu)$，以及往返于 p 和 d 的绕行（虚线）。

6 普通插入、普通配对插入和下一停靠点之前的取件插入

本节讨论普通、普通配对和在下一个停靠点之前接客 插入（见图 1）。在上述三种插入类型中，取件 p 和下车点d 都会插入到路线 $R(\nu)$上两个现有停靠点之间。要计算这几种类型中任意一种插入的成本，我们需要知道现有车辆停靠点与接送地点之间的距离。已有研究表明，采用椭圆剪枝的 BCH 搜索（见第 3.2 节）能够高效地计算这些距离 [4]。我们将其称为椭圆*BCH*搜索。此外，计算配对插入的成本还需要上车点至下车点距离 $\delta_{veh}(p,\ d)$。本节将说明如何扩展所需的距离查询，以支持多个接送地点。

6.1 椭圆 BCH 搜索

我们可以通过针对每个接送地点重复查询，轻松地将椭圆 BCH 查询扩展到多个接送地点。不过，即使采用椭圆剪枝，当接送地点数量较多时，运行时间仍可能变得难以接受。因此，为了提升算法在更多接送地点情况下的可扩展性，我们在椭圆 BCH 搜索的基础上补充了两种技术。我们仅针对接客点介绍这些技术，但它们同样适用于下客点。

使用排序桶的椭圆 BCH 搜索。首先，我们提出使用排序桶来减少 BCH 查询扫描的桶条目数量。我们仅针对源桶说明其原理，但同样可以将其应用于目标桶。

回顾一下，车辆 ν 的现有乘客约束为任意两个连续车辆停靠点 s_i 和 s_{i+1} 之间的绕行定义了一个余量 $\lambda(s_i, s_{i+1})$（见第 3.2 节）。当针对接客点 p 进行椭圆 BCH 搜索并扫描源桶条目 $(s, \delta^{\uparrow}(s_i, v)) \in B^{\uparrow}(v)$ 时，暂定距离 $\delta^{\uparrow}(s_i, v) + \delta^{\downarrow}(v, p)$ 只有在满足 $\delta^{\uparrow}(s_i, v) + \delta^{\downarrow}(v, p) \le \lambda(s_i, s_{i+1})$ 的情况下，才可能实现满足所有硬约束的插入。这意味着，只有当 $v(p) \le \lambda(s_i, s_{i+1}) - \delta^{\uparrow}(s_i, v)$ 时，该条目才与 p 相关。我们将 $\lambda_{res}(s_i, v) := \lambda(s_i, s_{i+1}) - \delta^{\uparrow}(s_i, v)$ 称为顶点 v处源桶条目 $(s_i, \delta^{\uparrow}(s_i, v))$ 的剩余余量。

我们按照剩余余量递减的顺序，对每个顶点 v 处每个源桶中的条目进行排序。随后，针对取件点 p 的椭圆BCH搜索可以在扫描到

v 某个条目 $(s, \delta^{\uparrow}(s, v))$ 后停止扫描，且该条目满足 $\delta^{\downarrow}(v, p) > \lambda_{res}(s, v)$。在这种情况下，对于任何后续条目 $(s', \delta^{\uparrow}(s', v))$，我们都有$\delta^{\downarrow}(v, p) > \lambda_{res}(s, v) \geq \lambda_{res}(s', v)$，因此剩余条目无法产生任何符合硬约束的插入。维护每个桶的顺序会在插入桶条目时产生时间开销。不过，由于桶的大小较小，这种开销是有限的。请注意，在只有一个取件点的情况下，也可以应用排序桶。

捆绑式椭圆 BCH 搜索。 其次，我们采用捆绑式椭圆 *BCH* 搜索 ，利用接客点的局部性。

与任何捆绑式搜索一样，捆绑式椭圆 BCH 搜索以 k 接客点为起点，并在每次边松弛时更新k 距离（参见第 3.1 节）。此外，我们还可以捆绑桶条目扫描。每当扫描某个停靠点 s 对应的桶条目时，搜索就会尝试改进 s 与任一 k 接客点之间的任意 k 暂定距离。由于位置相近的接客点共享相似的收缩层次搜索空间，我们可以有效地捆绑椭圆 BCH 搜索中的边松弛和桶条目扫描。此外，我们可以使用向量化指令并行执行边松弛和桶条目扫描。与此同时，仍可应用椭圆剪枝和排序桶。据我们所知，我们的算法是首个明确使用捆绑式 BCH 搜索的算法。这个想法源自 [3]中使用的捆绑式收缩层次搜索。

6.2 PD 距离搜索

计算上车点至下车点距离，即接客点与下车点之间的距离，是一个多对多最短路径问题，其中源点集和目标点集均为局部集合。

我们的算法采用 BCH 方法来解决这个问题。我们为所有下车点在其反向收缩层次搜索空间中生成桶条目。然后，以每个接客点为根节点，在向上的收缩层次搜索图中运行查询，利用下车点的桶条目查找上车点至下车点距离。我们提出了两种改进这些 BCH 搜索的方法。

首先，令 $\delta_{PD}^{\max}$为所有上车点至下车点距离的上界。这样，我们只需在半径为 $\delta_{PD}^{\max}$ 的范围内生成并扫描桶条目。我们使用

$$\delta_{PD}^{\max} := \max_{p \in P_\rho} \delta_{veh}(p, orig) + \delta_{veh}(orig, dest) + \max_{d \in D_\rho} \delta_{veh}(dest, d).$$

我们可以分别以起点 和终点为根节点，通过两次局部 Dijkstra 搜索，计算所有 $p \in P_\rho$ 对应的 $\delta_{veh}(p,$ 起点$)$，以及所有 $d \in D_\rho$ 对应的 $\delta_{veh}($终点 $d)$。通过一次预备的收缩层次查询，我们可以得到 $\delta_{veh}($起点，终点$)$。

其次，我们可以再次使用捆绑 BCH 搜索。更具体地说，我们可以为批次中的 k 下车点生成桶条目，然后为批次中的 k 接客点运行查询，其中k 是一个配置参数。同样，捆绑的上车点至下车点距离搜索利用了接客点和下车点的局部性，使我们能够采用 SIMD 并行。

6.3 枚举普通插入、普通配对插入和下一停靠点前接客插入

运行椭圆 BCH 查询和上车点至下车点距离搜索后，我们就知道了普通插入和普通配对 插入所需的所有距离。对于椭圆 BCH 查询 [4]找到的一组候选车辆，我们使用 $0 < i \leq j < k(\nu)$枚举插入项$\iota = (r, p, d, \nu, i, j)$。我们计算每种插入的成本 $c(\iota)$，并在 $c(\iota) < c(\iota^*)$ 时将 ι^* 更新为 ι 。

要计算下一个停靠点之前取件（PBNS）插入 $\iota = (r, p, d, \nu, 0, j)$ 的成本，我们缺少从车辆当前位置 $l_c(\nu)$ 到距离 $\delta_{veh}(l_c(\nu), p)$ 的知识，直到

取件 p。LOUD 建议根据通过考虑 $\delta_{veh}(s_0, p)$ 而非 $\delta_{veh}(s_0, l_c(\nu)) + \delta_{veh}(l_c(\nu)\ p)$ 所获得的 ι 成本下界，对接客绕行 [4] 的 PBNS 插入进行筛选。该筛选器会丢弃几乎所有 PBNS 插入。对于剩余的 PBNS 插入，我们采用基于桶的方法计算缺失的距离 $\delta_{veh}(l_c(\nu)\ p)$。我们为每辆受影响车辆的当前位置生成源桶条目，并从各接客点执行捆绑查询。每个请求的此类查询平均数量少于 0.5。

7 最后一站后接客插入

在本节中，我们考虑最后一站后接客 （PALS）插入。PALS 插入的主要挑战是计算从最后一站到接客点的距离。LOUD 的作者发现，椭圆剪枝不适用于这些距离的计算 [4]。因此，LOUD 使用以$orig$为根的反向 Dijkstra 搜索；当搜索无法再找到优于当前已知最佳方案的插入时，便提前停止。对于多个接客点，我们可以针对每个接客点分别运行反向 Dijkstra 搜索，以计算所需的距离。这些 Dijkstra 搜索也可以合并执行，以利用接客点的局部性。

然而，即使只有一个接客点，这次 Dijkstra 搜索也占用了 LOUD 算法运行时间的很大一部分。因此，对于大量接客点，我们预计运行时间将难以接受。在本节中，我们介绍两种用于计算最后一站距离的新型基于桶收缩层次的方法。在本节余下部分，令 $\hat{c}$ 表示当前已知最佳插入成本的上界（初始为 $\hat{c} := c(\iota^*)$ 。

PALS 插入的成本函数重述。请注意，任何 PALS 插入 $\iota = (r,\ p,\ d,\ \nu,\ k(\nu)\ k(\nu))$ 的成本完全由取件点 p、上车点至下车点距离$\delta_{veh}(p,\ d)$、步行距离 $\delta_{psg}(d,\ \text{目的地})$、 ν在 $s_{k(\nu)}$的出发时间 $t_{dep}^{min}(s_{k(\nu)})$，以及最后一站距离 $\delta_{veh}(s_{k(\nu)},\ p)$决定。因此，我们可以将 ι 的成本写作

$$c(\iota) = c'(r,\ p,\ \delta_{veh}(p,\ d)\ \delta_{psg}(d,\ \text{目的地})\ t_{dep}^{min}(s_{k(\nu)})\ \delta_{veh}(s_{k(\nu)},\ p)).$$

7.1 面向 PALS 的末站 BCH 搜索

虽然椭圆剪枝不适用，但我们仍可以采用 BCH 搜索方法，计算从最后一站到接客点的距离。为此，我们为每个 $v \in V$维护一个最后一站桶 $B^{\uparrow}\,\text{last}(v)$。对于每个最后一站 $s_{k(\nu)}$，我们在以 $s_{k(\nu)}$为根的 CH 上行搜索空间中的每个顶点 v 处，都生成一个条目 $(s_{k(\nu)}, \delta^{\uparrow}(s_{k(\nu)}, v))$ $\in B^{\uparrow}\,\text{last}(v)$。然后，对于每个接客点$p \in P_\rho$，我们在 $G_p^{\downarrow}$ 中执行一次单独的（最后一站）BCH 查询，扫描每个已确定顶点处的最后一站桶，以计算从最后一站到 p的最短路径距离。当搜索扫描到条目 $(s_{k(\nu)}, \delta^{\uparrow}(s_{k(\nu)}, v))\ \in B^{\uparrow}\,\text{last}(v)$ 时，会尝试用 $\delta^{\uparrow}(s_{k(\nu)}, v)\ + \delta^{\downarrow}(v, p)$ 改进暂定距离$\delta(s_{k(\nu)}, p)$。最终，将为每个最后一站 $s_{k(\nu)}$找到最短距离 $\delta_{veh}(s_{k(\nu)}, p)$。

每当车辆的最后一站发生变化时，我们都会运行两次正向 CH 搜索，以移除旧最后一站的桶条目，并为新的最后一站插入新条目。当当前距离已无法通过 PALS 插入获得低于已知最优成本的成本时，我们会提前停止搜索。此外，我们还可以像捆绑式椭圆 BCH 搜索（参见第6.1节）那样捆绑 BCH 查询并使用 SIMD 并行。

使用排序桶对桶扫描进行成本剪枝。这种方法仍存在一个问题：最后一站桶的规模。在没有椭圆剪枝的情况下，桶中包含的

条目会多得多，尤其是在收缩层次中秩较高的顶点处。因此，查询必须扫描大量桶条目，导致最后一站 BCH 方法失效。

[4] 的未来工作部分建议对每个最后一站桶中的条目进行排序，以解决此问题。对于每个 $v \in V$，我们按距离 $\delta^{\uparrow}(s_{k(\nu)}, v)$ 的升序对 $B^{\uparrow}_{\text{最后}}(v)$ 中的条目进行排序。假设一个以 $p \in P_\rho$ 为根的取件查询扫描桶 $B^{\uparrow}_{\text{最后}}(v)$。令

$$c_{\min}(x) := c'(r, p, \delta^{\min}_{\text{pd}}, 0, t_{req}(r), x + \delta^{\downarrow}(v, p)) \quad \text{where } \delta^{\min}_{\text{pd}} := \min_{p \in P_\rho, d \in D_\rho} \delta_{veh}(p, d).$$

然后，对于任意条目 $e = (s_{k(\nu)}, \delta^{\uparrow}(s_{k(\nu)}, v)) \in B^{\uparrow}\text{last}(v)$，值 $c_{\min}(\delta^{\uparrow}(s_{k(\nu)}, v))$ 是一种与车辆无关的下界，适用于任何行驶距离至少为 $\delta^{\uparrow}(s_{k(\nu)}, v) + \delta^{\downarrow}(v, p)$ 至 p 的 PALS 插入的成本。注意，$c_{\min}(x)$ 关于 x 是线性的。因此，由于 $B^{\uparrow}\text{last}(v)$ 的条目按 $\delta^{\downarrow}(s_{k(\nu)}, v)$ 排序，如果$c_{\min}(\delta^{\uparrow}(s_{k(\nu)}, v)) > \hat{c}$，我们就可以停止桶扫描。

更新成本上界。对于成本剪枝所需的成本上界$\hat{c}$，可以直接使用 $c(\iota^*)$。不过，我们也可以在搜索过程中更新$\hat{c}$。每个找到的暂定距离 $\delta(s_{k(\nu)}, p)$ 都是实际最短距离 $\delta_{veh}(s_{k(\nu)}, p)$ 的一个上界。因此，每当更新暂定距离 $\delta(s_{k(\nu)}, p)$ 时，我们都可以计算一个上界

$$c_{\max} = c'(r, p, \delta_{veh}(p, dest), 0, t^{min}_{dep}(s_{k(\nu)}), \delta(s_{k(\nu)}, p))$$

关于包含 ν 和 p 的最佳 PALS 插入的成本。如果 $c_{\max} < \hat{c}$，我们将$\hat{c}$ 更新为 $c_{\max}$。此技术可以提前找到近似的成本上界，这有助于桶扫描的停靠条件。

7.2 PALS 的集体最后停靠点搜索

最后，我们提出一种搜索方法，其基本思路是：实际上，我们并不需要知道每个最后一站与每个接客点之间的距离。如果我们预先知道最佳 PALS 插入 $\iota^*_{\text{pals}} = (r, p, d, \nu, k(\nu), k(\nu))$，就只需要找到 $\delta_{veh}(s_{k(\nu)}, p)$。显然，我们无法预先知道 ι^*_{pals}，但我们发现，可以通过相互比较各个取件点（或者实际上是各个 PD 对），来剪枝针对单个取件点的距离查询。我们介绍一种集体 BCH 查询，它可以找到最佳 PALS 插入 ι^*_{pals} 以及最后一站距离 $\delta_{veh}(s_{k(\nu)}, p)$。下文将说明，表示 PD 对的标签如何在 CH 搜索图中传播，以及如何根据标签支配关系对这些标签进行剪枝。

开放标签与关闭标签。顶点 $v \in V$处的一个 PD 对标签（p，d，$\delta^{\downarrow}(v, p)$）由取件点 $p \in P_\rho$、下车点 $d \in D_\rho$和向下距离 $\delta^{\downarrow}(v, p)$组成。在每个顶点 $v \in V$处，都有一组开放标签开放(v)和一组关闭标签关闭(v)。开放标签是指尚未确定的标签。对于每个开放标签 $l = (p, d, \delta^{\downarrow}(v, p))$，我们会存储一个下界$c_{\min}(l)$，用于表示在 $G^{\downarrow}_v$中为 l 找到的 PALS 插入的成本

$$c_{\min}(l) := c'(r, p, \delta_{veh}(p, d), \delta_{psg}(d, dest), t_{req}(r), \delta^{\downarrow}(v, p))$$

算法概述。我们在算法 1 中给出了集体 BCH 搜索的伪代码。搜索维护一个优先队列 Q，其中包含所有开放标签，并按$c_{\min}$递增排序。初始时，对于每个取件点 $p \in P_\rho$，都会为每个$d \in D_\rho$创建一个开放标签 $(p, d, 0) \in open(p)$。只要 Q 中存在一个标签 l，满足 $c_{\min}(l) \leq \hat{c}$，且已知任意插入的成本上界$\hat{c}$，搜索就会继续执行下一步。在搜索的每一步中，都会从 Q 中移除标签 $l := \min(Q)$，并按照下文所述将其确定。

算法 1 用于查找从最后一站到取件地点距离的集体 BCH 搜索。

过程 CollectiveBCH(P_ρ, D_ρ, $G^\downarrow = (V^\downarrow, E^\downarrow)$) : ($p^*, d^*$) and $\delta_{veh}(s_{k(\nu)}, p^*)$
  $Q :=$ 包含 $key_Q(l) = c_{\min}(l)$ 的标签 PQ；开放(v) := ∅, 已关闭(v) := ∅ 对于 $v \in V$
  $\hat{c} := c(\iota^*)$
  对于每个 $(p, d) \in P_\rho \times D_\rho$ **执行**
    insertLabelAtVertex(p, (p, d, 0))
  当 $Q \neq \emptyset$ **时 do**
    $l :=$ Q.deleteMin()
    如果 $c_{\min}(l) > \hat{c}$ **则返回**
    settleLabel(l)

过程 SettleLabel($l = (p, d, \delta^\downarrow(v, p))$)
  开放（v）。remove（l）；关闭（v）。insert（l）　▷ 标记 l 关闭
  对于每个 $e = (s_{k(\nu)}, \delta^\uparrow(s_{k(\nu)}, v)) \in B^\uparrow{}_{\text{最后}}(v)$ **执行**　▷ 扫描 v 处的最后一站条目，查找 l
    如果 $c_{\min}(l, e) > \hat{c}$ **则中断**
    如果 $c_{\max}(l, e) < \hat{c}$ **则**
      (p^*, d^*) := (p, d); $\hat{c} := c_{\max}(l, e)$
  对每个 $(u, v) \in E^\downarrow$ **执行**　▷ 将 l 传播给 v 的邻居
    insertLabelAtVertex(u, ($p, d, \ell^+(u, v) + \delta^\downarrow(v, p)$))

过程 InsertLabelAtVertex(v, $l' = (p, d, \delta^\downarrow(v, p))$)
  如果 $c_{\min}(l') > \hat{c}$ **则返回**
  对每个 $l \in$ 开放(v) ∪ 已关闭(v) **执行**
    如果 l 支配 l'，**则返回**
  对每个 $l \in$开放(v) **执行**
    如果 l' 支配 l，**则** 开放(v).remove(l)
  $open(v)$.insert(l'); Q.insert(l')

确定开放标签。 确定开放标签 $l = (p, d, \delta^\downarrow(v, p))$ 包含三个步骤：首先，在 v 将 l 标记为已关闭，即将其从 $open(v)$ 移至 $closed(v)$。其次，通过遍历最后一站桶 $B^\uparrow{}_{last}(v)$ 中的所有条目，搜索新的最佳插入。对于 $B^\uparrow{}_{last}(v)$ 中的每个条目 $e = (s_{k(\nu)}, \delta^\uparrow(s_{k(\nu)}, v))$，我们计算一个成本上界

$$c_{\max}(l, e) := c'(r, p, \delta_{veh}(p, d), \delta_{psg}(d, dest), t^{min}_{dep}(s_{k(\nu)}), \delta^\uparrow(s_{k(\nu)}, v) + \delta^\downarrow(v, p)).$$

如果 $c_{\max}(l, e) < \hat{c}$，我们将 $\iota = (r, p, d, \nu, k(\nu)\ k(\nu))$ 标记为当前已知的最佳 PALS 插入，存储暂定距离 $\delta(s_{k(\nu)}, p) = \delta^\uparrow(s_{k(\nu)}, v) + \delta^\downarrow(v, p)$，并更新 $\hat{c} := c_{\max}(l, e)$。注意，如果 $\delta(s_{k(\nu)}, p)$ 是最短路径距离，则 $c_{\max}(l, e)$ 是 PALS 插入 $\iota = (r, p, d, \nu, k(\nu)\ k(\nu))$ 的精确成本。由于 BCH 搜索会找到最短上行-下行路径，我们最终会找到最佳 PALS 插入。与之前一样，我们可以提前终止每次桶扫描。为此，我们计算与车辆无关的成本下界 $c_{\min}(l, e)$，使得当 $c_{\min}(l, e) > \hat{c}$ 时，可以提前终止搜索

$$c_{\min}(l, e) := c'(r, p, \delta_{veh}(p, d), \delta_{psg}(d, dest), t_{req}(r), \delta^\uparrow(s_{k(\nu)}, v) + \delta^\downarrow(v, p)).$$

第三，我们将 l 传播到 v的所有相邻顶点。对于每个邻接顶点 $w \in V$，若其对应的 $(w, v) \in G^\downarrow$，我们会在 w处创建一个新的开放标签 $l' = (p, d, \ell^+(w, v) + \delta^\downarrow(v, p))$。这里，我们

通过成本剪枝丢弃 l'：如果该 PD 对在此距离下的下界成本 $c_{\min}(l')$ 超过 $\hat{c}$，就丢弃它。此外，如果 l' 在 v 处被 v 处的现有标签支配，我们也可以对其进行剪枝，如下所述。

支配剪枝。对于每个 PD 对，将一个标签传播遍整个搜索空间的成本过高。然而，我们发现可以比较同一顶点处的标签，并采用一种称为支配剪枝的技术剪除受支配的标签。直观地说，如果我们知道在顶点 v 处找到的任何插入中，使用 l' 的插入成本都高于使用 l的等效插入，那么标签 l 就支配标签 l'。

为使其形式化，我们首先为标签 l在$G_v^{\downarrow}$中找到的 PALS 插入的成本定义一个上界。设 $l=(p,\ d,\ \delta^{\downarrow}(v,\ p))$ 是位于 v处的一个标签，而 $e=(s_{k(\nu)},\ \delta^{\uparrow}(s_{k(\nu)},\ w)) \in B^{\uparrow}{}_{\text{最后}}(w)$ 是顶点 $w \in V_v^{\downarrow}$处的一个最后一站桶条目。于是，我们将 ν 从$s_{k(\nu)}$ 经由 w 行驶到 p 并 v 的插入成本上界定义为

$$c_{\max}(l,v,e) := c'(r,p,\delta_{veh}(p,d),\delta_{psg}(d,dest),t_{dep}^{min}(s_{k(\nu)}),\delta^{\uparrow}(s_{k(\nu)},w)+\delta^{\downarrow}(w,v)+\delta^{\downarrow}(v,p))$$

据此，我们可以正式定义标签之间的支配关系：

▶ **定义 6.** 在顶点 $v \in V$ 处，*PD* 对标签 l 支配 另一个标签 l'，当且仅当对于每个 $w \in V_v^{\downarrow}$，以及 $e \in B^{\uparrow}{}_{\text{最后一个}}(w)$，$c_{\max}(l,v,e) < c_{\max}(l',v,e)$。

▶**定理 7.** 如果标签 l 在 v处支配另一个标签 l'，我们就不需要在 v处确定 l'。

证明。可以通过反证法证明这一点。假设 $l=(p,d,\delta^{\downarrow}(v,p))$ 在 v处支配 $l'=(p',d',\delta^{\downarrow}(v,p'))$。进一步假设 $\iota=(r,p',d',\nu,k(\nu)\ k(\nu))$ 是最佳的 PALS 插入。令 π 为从$s_{k(\nu)}$ 到 p'的一条最短路径。不失一般性，π 是收缩层次中的一条上升-下降路径，由向上的前缀 $\pi^{\uparrow}$ 和向下的后缀 $\pi^{\downarrow}$ 组成。如果 $\pi^{\downarrow}$ 不包含 v，那么集体搜索将无法在 $G_v^{\downarrow}$中找到 π，我们也就不必在 v处确定 l'。

否则，$\pi^{\downarrow}=(w,\ldots,v,\ldots,p')$ 与 $w \in V_v^{\downarrow}$。令 $e=(s_{k(\nu)},\delta^{\uparrow}(s_{k(\nu)},w)) \in B^{\uparrow}{}_{\text{最后}}(w)$。由于 π 是一条最短路径，我们知道

$$\begin{aligned} c_{\max}(l',v,e) &= c'(r,p',\delta_{veh}(p',d'),\delta_{psg}(d',dest),t_{dep}^{min}(s_{k(\nu)}),\delta_{veh}(s_{k(\nu)},p')) \\ &= c((r,p',d',\nu,k(\nu),k(\nu))). \end{aligned}$$

然而，l 支配 l'，这意味着 that

$$c((r,p,d,\nu,k(\nu),k(\nu))) \le c_{\max}(l,v,e) < c_{\max}(l',v,e) = c((r,p',d',\nu,k(\nu),k(\nu)))$$

这与 ι 是最佳 PALS 插入相矛盾。因此，我们无需在 v 确定标签 l'，即可找到 ν的最佳配对。◀

高效计算支配关系。我们发现，由于成本函数具有非线性，高效计算支配关系并非易事。

考虑为 $i=1$ 选择的两个标签 $l_i=(p_i,d_i,\delta^{\downarrow}(v,p_i))$，以及在 $G_v^{\downarrow}$ 中为这些标签找到的两个 PALS 插入 $\iota_i=(r,p_i,d_i,\nu,k(\nu)\ k(\nu))$。假设车辆 ν 在某个时间 $t=t_{dep}^{min}(s_{k(\nu)})+\delta_{veh}(s_{k(\nu)},v)$ 到达 v。如果我们知道 t，就能确定成本差 $c(\iota_1)-c(\iota_2)$，而不受实际车辆 ν 的影响。换言之，在 $G_v^{\downarrow}$ 中为 l_1 和 l_2 找到的任意插入之间的成本差 $\Delta_c(l_1,l_2,t)$ 是 t 的函数。因此，如果对于所有 $t \ge 0$，$\Delta_c(l_1,l_2,t) < 0$，那么在 $G_v^{\downarrow}$ 中找到的每个插入使用 l_1 都会比使用 l_2更优，即 l_1 支配 l_2。

如果 PALS 插入的成本函数随 t线性增长，那么 $\Delta_c(l_1,l_2,t)$ 相对于 t将为常数。在这种情况下，我们只需使用$\Delta_c(l_1,l_2,0) < 0$检查支配关系即可。然而，成本函数并不随 t线性增长：首先，

只要车辆早于乘客到达 p_i ，成本相对于 t 就是常数，即 $t+\delta^{\downarrow}(v,\ p_i)+t^{min}_{stop}\le t_{req}(r)+\delta_{psg}(orig\ p_i)$（参见第4节.2）。其次，由于等待时间软约束和行程时间软约束，等待时间和行程时间超过特定阈值后，成本函数中会加入线性惩罚项，而这两者都会对其产生贡献。

不同标签对应的乘客到达时间 p_i 以及软约束惩罚项的阈值各不相同。因此， $\Delta_c(l_1,\ l_2,\ t)$ 会随 t变化。由于我们不知道在 $G_v^{\downarrow}$中找到的插入可能对应 t 的哪些取值，因此无法直接判断 l_1 是否支配 l_2。

近似支配关系。相反，我们通过计算一个充分前置条件，对支配关系进行下近似。为此，我们求出上界$\Delta_c^{\max}(l_1,\ l_2)\ge\max_{t\ge0}\Delta_c(l_1,\ l_2,\ t)$，用于限定在 $G_v^{\downarrow}$中为 l_1 和 l_2 找到的任意插入之间的插入成本差异。然后，若 $\Delta_c^{\max}(l_1,\ l_2)<0$，则 l_1 支配 l_2 。

现在我们说明如何求出这个上界：在 p_1 和 p_2 处的出发时间差的最大值为 $t^{max}_{dep}(l_1)-t^{min}_{dep}(l_2)$，其中

$$t^{max}_{dep}(l_1):=\max\{\delta^{\downarrow}(v,p_1)+t^{min}_{stop},\delta_{psg}(p_1,orig)\}$$
$$t^{min}_{dep}(l_2):=\delta^{\downarrow}(v,p_2)+t^{min}_{stop}$$

然后，对于在 $G_v^{\downarrow}$中找到的每个插入， l_1 和 l_2 之间的绕行差异都有上界 $\Delta_{detour}(l_1,\ l_2):=t^{max}_{dep}(l_1)+\delta_{veh}(p_1,\ d_1)-t^{min}_{dep}(l_2)-\delta_{veh}(p_2,\ d_2)$。对于行程时间差异，我们首先定义：

$$t^{max}_{arr}(l_1):=t^{max}_{dep}(l_1)+\delta_{veh}(p_1,d_1)+\delta_{psg}(d_1,dest)$$
$$t^{min}_{arr}(l_2):=t^{min}_{dep}(l_2)+\delta_{veh}(p_2,d_2)+\delta_{psg}(d_2,dest)$$

然后，行程时间差异的上界为 $\Delta_{trip}(l_1,\ l_2):=t^{max}_{arr}(l_1)-t^{min}_{arr}(l_2)$。我们将等待和行程时间软约束的惩罚项差异的上界定义为

$$\Delta^{vio}_{wait}(l_1,l_2):=\gamma_{wait}\max\{t^{max}_{dep}(l_1)-t^{min}_{dep}(l_2),0\}$$
$$\Delta^{vio}_{trip}(l_1,l_2):=\gamma_{trip}\max\{\Delta_{trip}(l_1,l_2),0\}$$

请注意，绕行和行程时间的差异可以为负，以表示 l_1 具有成本优势，但惩罚项的差异不能为负。即使 $t^{max}_{dep}(l_1)<t^{min}_{dep}(l_2)$或$\Delta_{trip}(l_1,\ l_2)<0$，我们也可能在 $G_v^{\downarrow}$ 中找到某些插入，此时两个标签都不适用任何惩罚项。在这种情况下，惩罚项差异必须为零。令 $\Delta_{walk}(l_1,\ l_2)$表示步行成本的固定差异。综合以上各项，可得

$$\Delta_c^{\max}(l_1,l_2):=\Delta_{detour}(l_1,l_2)+\tau\Delta_{trip}(l_1,l_2)+\omega\Delta_{walk}(l_1,l_2)+\Delta^{vio}_{wait}(l_1,l_2)+\Delta^{vio}_{trip}(l_1,l_2)$$

我们可以利用在 v 已知的信息，在不查看搜索树后续内容的情况下，以常数时间计算 $\Delta_c^{\max}(l_1,\ l_2)$。由于我们对支配关系进行了低估，因此可能存在 l_1 实际上支配 l_2 ，但我们的条件并不成立。不过，我们发现，我们的支配判据仍能在早期剪枝掉绝大多数标签。

局限性。 需要指出的是，集体搜索找到的插入 ι 只有在 ι 满足服务时间硬约束时，才能保证是最佳的 PALS 插入。由于我们的搜索忽略了服务时间约束，即使存在其他不违反该约束的 PALS插入，它也可能返回一个违反约束的插入。

因此，如果 ι 违反服务时间约束，我们就退而采用单个最后一站 BCH 搜索，计算每个最后一站到每个取件点的距离。在集体搜索过程中找到的良好成本上界，也可用于这些后备的单个 BCH 搜索。我们发现，只有在极其罕见的情况下才需要这样做。

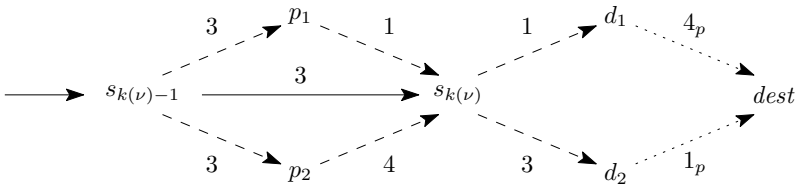


图2 示例：在最后一站之后插入下车点，说明为何下车点之间的支配关系是偏序关系。图中展示了现有车辆路线（实线），包括当前最后两个停靠点（$s_{k(\nu)-1}$、$s_{k(\nu)}$）、这两个停靠点之间的两个接客点（p_1、p_2），以及最后一站之后的两个下车点（d_1、d_2）和必要的绕行（虚线），还显示了从任一下车点到目的地的步行距离（点线）。边上标注了车辆行驶时间或乘客行程时间（以 p 标记）。

8 最后一站之后的下车点插入

最后一种插入类型是最后一站之后的下车点 (DALS) 插入（见图 1）。与 PALS 插入类似，我们面临的问题是，不知道车辆最后一站到下车点的距离。我们可以采用相同的 Dijkstra 搜索方法，以及单个和集体 BCH 搜索来求得这些距离，但有一些细微差异。

首先，与 PALS 情况相比，成本剪枝的效果较差，因为成本下界无法纳入上车点至下车点距离。其次，在 DALS 情况下，由于缺乏有关在路线较早位置插入取件的成本信息，我们无法在搜索过程中更新全局成本上界 $\hat{c}$。第三，集体 BCH 搜索在 PALS 和 DALS 情况下还存在一些更复杂的差异。本节余下部分将进一步详细讨论这些差异。

帕累托最优下车点。 与 PALS 情况相比，DALS 情况下集体搜索最大的区别在于搜索目标。集体 PALS 搜索总是会找到一个最佳 PD 对，而我们无法找到一个最佳下车点。相反，我们会为每辆车辆找到一组帕累托最优下车点。

要理解为什么车辆 $\nu \in F$ 不一定总能有一个最佳下车点，假设我们想判断 $d_1 \in D_\rho$ 是否总能产生优于 $d_2 \in D_\rho$ 的 DALS 插入。如果对于任何 DALS 插入 $\iota_1 = (r,\ p,\ d_1,\ \nu,\ i,\ k(\nu))$，以及任意取件 $p \in P_\rho$ 和 $0 \le i < k(\nu)$，下车点 d_1 对 ν 而言总是优于 d_2，则 $c(\iota_1) < c(\iota_2)$，且 $\iota_2 = (r,\ p,\ d_2,\ \nu,\ i,\ k(\nu))$。

考虑成本差异 $c(\iota_1) - c(\iota_2)$。在 $s_{k(\nu)}$ 之前，两种插入方式的车辆绕行和行程时间相同。因此，ι_1 与 ι_2 在绕行、行程时间、现有乘客增加的行程时间、步行时间以及等待时间约束惩罚项方面的差异，完全独立于 p 和 i的选择。然而，对于 p 和 i的不同选择，任一种插入方式都可能违反行程时间软约束，也可能不违反。因此，在某些 p 和 i 的选择下，ι_1 可能更优；而在另一些选择下，ι_2 可能更优。

为说明这一问题，请看图2所示的示例。假设 $\tau = 1$ 和$\gamma_{trip} = 10$。此外，令最长行程时间 $t^{max}_{arr}(r) - t_{req}(r) = 10$。令 δ_{psg}(原始 p_1) < 3，以及 δ_{psg}(原始 p_2) < 3。

考虑针对 $i = 1,\ 2$ 的插入 $\iota_{1i} = (r,\ p_1,\ d_i,\ \nu,\ k(\nu) - 1,\ k(\nu))$。这两种插入都没有违反行程时间软约束，因为 $t_{trip}(\iota_{11}) = 3 + 1 + 1 + 4 = 9$ 和 $t_{trip}(\iota_{12}) = 3 + 1 + 3 + 1 = 8$。绕行差异为 $t_{detour}(\iota_{11}) - t_{detour}(\iota_{12}) = 5 - 7 = -2$，导致成本差异为 $c(\iota_{11}) - c(\iota_{12}) = -1$。因此，对于 p_1，d_1 是更好的选择。

现在，考虑针对 $i = 1,\ 2$ 的插入方案 $\iota_{2i} = (r,\ p_2,\ d_i,\ \nu,\ k(\nu) - 1,\ k(\nu))$。这里，两种插入方案都违反了行程时间软约束，因为 $t_{trip}(\iota_{21}) = 12$ 且 $t_{trip}(\iota_{22}) = 11$。绕行和行程时间的差异与之前一样累加，但还要加上.....的差异

18 拼车场景下多次接送地点的快速多对多路径规划

表 1 基准实例的关键数据。

实例	$\|V\|$	$\|E\|$	#车辆	#req.
`Berlin-1pct`	73689	159039	1000	16569
`Berlin-10pct`	73689	159039	10000	149185
`Ruhr-1pct`	394049	840587	3000	49707
`Ruhr-10pct`	394049	840587	30000	447555

表 2 在 `Berlin-1pct`、 `Berlin-10pct`、 `Ruhr-1pct`和 `Ruhr-10pct` 实例中，特定步行半径 ρ 下的平均接客点数量（N_p）和下客点数量（N_d）。

	B-1%		B-10%		R-1%		R-10%	
ρ	N_p	N_d	N_p	N_d	N_p	N_d	N_p	N_d
0s	1	1	1	1	1	1	1	1
150s	10	10	10	10	12	11	12	11
300s	33	31	33	31	35	33	35	33
450s	68	66	68	67	70	68	70	68
600s	115	113	115	114	114	112	114	112

惩罚项，我们得到 $c(\iota_{21}) - c(\iota_{22}) = -1 + 10 = 9$。因此，对于 p_2， d_2 是更好的选择。

这表明，我们可能无法为DALS插入选择唯一的最佳下车点。相反，我们只能知道：如果下车点 d_1 在有行程时间违规和无行程时间违规的DALS插入中都优于下车点 d_2，那么 d_1 总能带来更优的插入。因此，对于每辆车，我们最终只能获得一组帕累托最优下车点。

算法概述与部分支配关系。我们的集体搜索会在每个带有 $d \in D_\rho$的顶点 $v \in V$处维护形如（d， $\delta^{\downarrow}(v,\ d)$）的开放和已关闭下车点标签。初始时，在每个 $d \in D_\rho$处创建一个开放标签（d，0）。与PALS情况一样，我们为每个开放标签 l关联一个下界成本 $c_{\min}(l)$。在每一步中，具有最小下界成本的开放标签 l 会通过在其顶点 v处关闭该标签、扫描 v处的最后一站桶，并将 l 传播到 $G_v^{\downarrow}$中的相邻顶点而得到处理。

我们对桶扫描和整个搜索过程采用基于成本的停止准则。当一个标签传播到新的顶点时，我们会应用支配剪枝。对于支配关系，我们会检验：在考虑和不考虑行程时间惩罚项的情况下，一个下车点是否始终优于另一个下车点：

▶ **定义 8**。 设 $l_i = (d_i, \delta^{\downarrow}(v, d_i))$ 对于 $i = 1,2$ 是位于 $v \in V$ 的两个下车点标签。设

$$\Delta_{detour}(l_1, l_2) := \delta^{\downarrow}(v, d_1) - \delta^{\downarrow}(v, d_2)$$
$$\Delta_{trip}(l_1, l_2) := \delta^{\downarrow}(v, d_1) + \delta_{psg}(d_1, dest) - \delta^{\downarrow}(v, d_1) - \delta_{psg}(d_2, dest)$$

则 d_1 支配 d_2，当且仅当

1. $\Delta_{detour}(l_1, l_2) + \tau\Delta_{trip}(l_1, l_2) < 0$ 和2.
$\Delta_{detour}(l_1, l_2) + (\tau + \gamma_{trip})\Delta_{trip}(l_1, l_2) < 0$

9 实验评估

我们的源代码采用 C++17 编写，并使用 `-O3` 通过 GCC 9.4 编译。我们通过严格使用断言（实验时禁用）来检查实现的正确性。我们在一台运行 Ubuntu 20.04 的机器上进行实验，该机器配备 512 GiB 内存

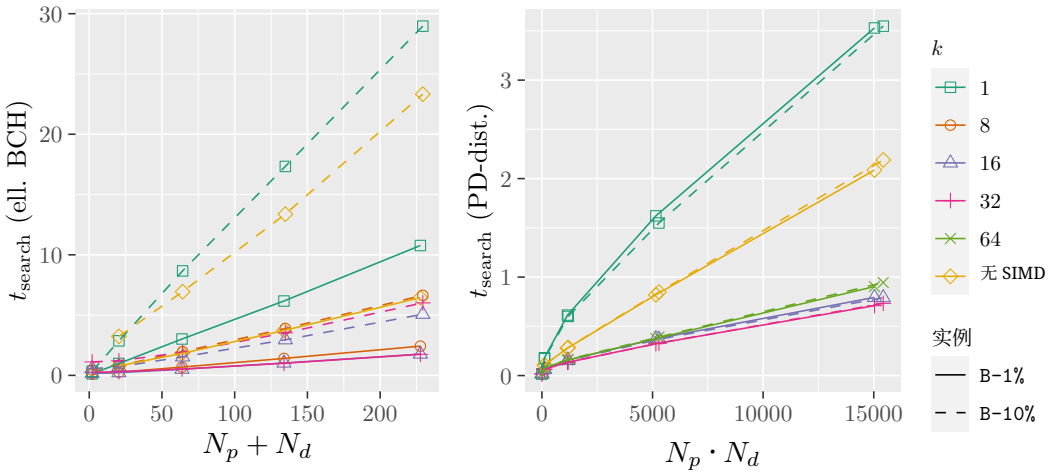


图 3 捆绑式椭圆 BCH 搜索（左）和 PD 距离搜索（右）的评估。展示了在 `Berlin-1pct` 和 `Berlin-10pct` 实例上执行相应搜索时的平均搜索时间（单位：毫秒），其中 $\rho \in \{0$ 秒、150 秒、300 秒、450 秒、600 秒$\}$。对于椭圆 BCH 搜索，考虑 $k \in \{1$、8、16、32$\}$；对于 PD 距离搜索，考虑 $k \in \{1$、16、32、64$\}$，并对 $k > 1$使用 SIMD 指令。此外，还展示了椭圆搜索在无 SIMD 指令、$k = 16$ 以及 PD 距离搜索在 $k = 32$下的运行时间。注意 x轴和 y轴的差异。

和两颗 2.3GHz 的 16 核 Intel Xeon E5-2670 v3 处理器。我们使用 32 位距离标签和具有 256 位寄存器的 AVX2 SIMD 指令集，通过一条向量指令最多计算 8 项操作。

我们在 `Berlin-1pct`（B-1%）、`Berlin-10pct`（B-10%）、`Ruhr-1pct`（R-1%）和 `Ruhr-10pct`（R-10%）请求集上评估我们的实现 [4]；这些请求集分别代表工作日柏林和莱茵-鲁尔大都市区拼车需求的 1% 和 10%。柏林的请求集基于 MATSim 交通仿真的开放柏林场景 [19] [12]人工生成。莱茵-鲁尔地区的请求集通过扩大柏林请求集 [4]获得。底层道路网络取自 OpenStreetMap 数据[1]。我们使用每条道路的已知限速来确定车辆网络中相应边的行驶时间。对于乘客网络，我们假设恒定的步行速度为 4.5 千米/小时。我们在表 1 中展示网络和请求集的规模。我们通过使用递增的步行半径 $\rho \in \{0$ 秒、150 秒、300 秒、450 秒、600 秒$\}$ 来扩大接客点 N_p 和下客点 N_d 的数量，这些半径对应于表 2 中给出的 PD 位置数量。除非另有说明，对于算法配置、输入和半径的每种组合，我们运行 5 次迭代 ρ 并报告平均运行时间。

对于模型参数，我们选择 $t_{wait}^{max} = 600$s、$t_{stop}^{min} = 60$s、$\gamma_{wait} = 1$、$\gamma_{trip} = 10$、$\tau = 1$ 和 $\omega = 0$。我们使用开源库 RoutingKit[2] 来计算道路网络所需的收缩层次，这对我们的实例来说只需几秒钟。对于任何输入实例中的每个图，这一预处理步骤都只需几秒钟。

20 拼车多次接送地点的快速多对多路径规划。

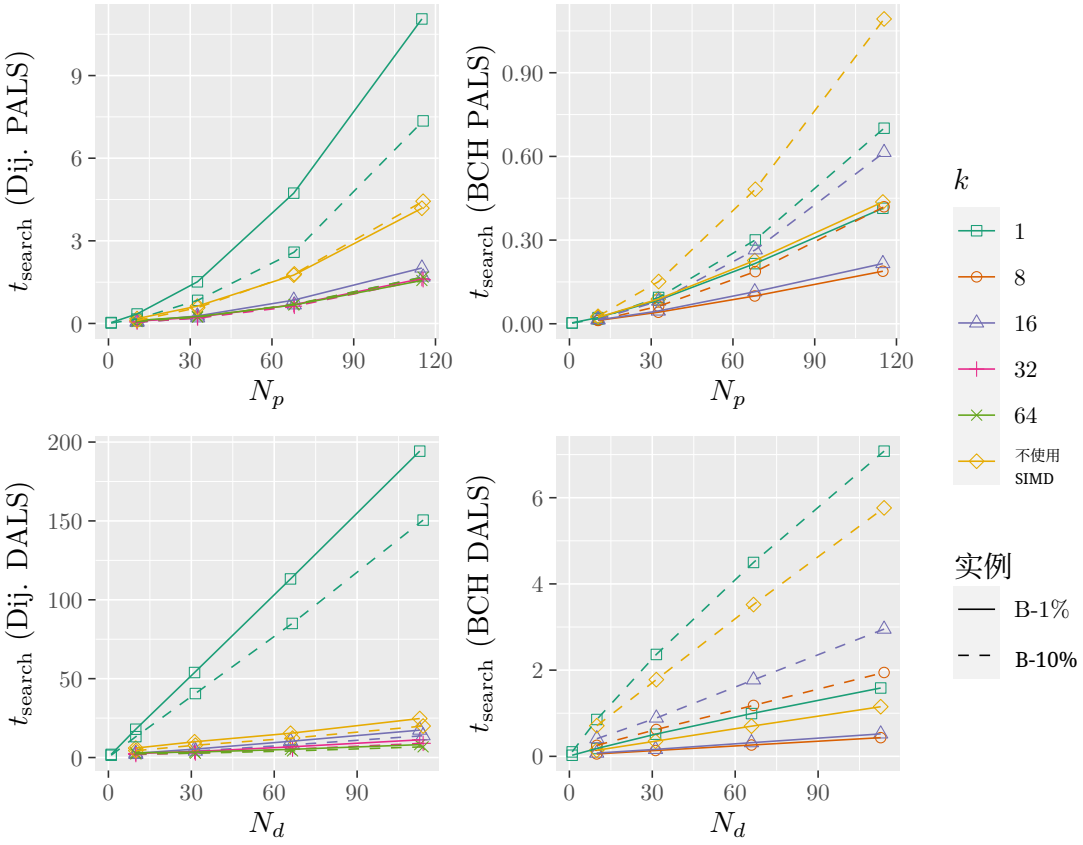


图 4 捆绑式末站搜索的评估。展示了在 PALS（上）和 DALS（下）情况下，对 `Berlin-1pct` 和 `Berlin-10pct` 实例使用 $\rho \in \{0s, 150s, 300s, 450s, 600s\}$时，捆绑式 Dijkstra 搜索（左）和单个 BCH 搜索（右）的平均搜索时间（毫秒）。Dijkstra 搜索考虑 $k \in \{1, 16、32, 64\}$，BCH 搜索考虑 $k \in \{1、8、16\}$，并对 $k > 1$ 使用 SIMD 指令。此外，还展示了在 Dijkstra 搜索中使用 $k = 64$ 、在 BCH 搜索中使用 $k = 8$ 时不使用 SIMD 指令的运行时间。请注意不同的 y 轴。

9.1 捆绑式搜索

在本节中，我们通过实验评估所述各类应用中的捆绑式搜索，并为每种应用找出 k 的最优值。我们使用`Berlin-1pct`和 `Berlin-10pct`实例进行实验，取值为 $\rho \in \{0s, 150s, 300s, 450s, 600s\}$。由于时间约束，我们先通过向量指令找出 k 的最优值，然后仅针对该值 k评估不使用向量指令时的运行时间。不过，我们指出， k 的最优值可能会因是否使用向量指令以及可用的 SIMD 指令集而异。由于 Dijkstra 搜索的运行时间过长，我们只执行一轮迭代。

捆绑式椭圆 BCH 搜索。 我们在图 3 中展示了使用 $k \in \{1, 8, 16, 32\}$ 时捆绑式椭圆 BCH 搜索的实验运行时间。我们发现， $k = 16$ 最适合此处。

[1] `https://download.geofabrik.de/`
[2] `https://github.com/RoutingKit/RoutingKit`

由于使用了捷径边，收缩层次中每个源点的搜索树只有在距离源点较远处才开始重叠。因此，我们可以在源点外围捆绑边松弛和桶扫描，但不能在更靠近每个源点的位置这样做。排序桶还会将工作重点进一步转移到靠近源点的位置，在那里， k 的较大值并不奏效。 $k = 16$ 的取值在两个方面之间实现了平衡：限制靠近源点处的开销，同时在更远处捆绑操作。

$k = 16$ 相较于 $k = 32$ 的优势在 `Berlin-10pct` 实例上更加明显。我们认为，这是因为车辆总数更多，这意味着桶平均包含更多条目。因此，对于排序桶而言，对靠近源点的工作部分的关注更加明显。

使用 $k = 16$ 并采用 SIMD 并行（无 *SIMD*）的捆绑式搜索仅带来小幅加速。同样，由于搜索工作中在距离源点更远处执行的部分所占比例更大，捆绑对于较小的实例效果更好。

捆绑式 PD 距离搜索。 在图 3 中，我们展示了使用 $k \in \{1、16、32、64\}$时，捆绑式 PD 距离搜索的运行时间。由于我们的 PD 距离 BCH 搜索相比椭圆 BCH 搜索会在距离源点更远处执行更多工作，因此所有搜索的搜索树重叠程度更高。这样一来，较大的 k 值便可以有效地捆绑边松弛，以及桶条目的生成和扫描。尽管如此，捆绑潜力仍然有限，而较大的 k 值会在靠近源点处带来额外开销。实际上，对于我们的 PD 距离搜索而言，$k = 32$ 是最佳选择。

此外，我们还考察了在 $k = 32$ 不使用 SIMD 指令时（不使用 *SIMD*）的运行时间。即使没有 SIMD 并行，我们观察到的加速比最高也达到 2.2。这同样是因为源点外围执行的工作量更大，而这部分工作很适合捆绑。

捆绑式末站搜索。 我们在图 4 中展示了 PALS 和 DALS 情况下，捆绑式 Dijkstra 搜索与单个 BCH 搜索的运行时间。

我们发现 Dijkstra 搜索很适合捆绑，因为在 $k = 64$下观察到的搜索时间最短。由于 Dijkstra 搜索不使用捷径边，每个源点对应的搜索会比 BCH 搜索更早相遇。因此，Dijkstra 搜索中数量庞大的边松弛操作绝大多数都可以很好地捆绑。即使不使用 SIMD 指令，捆绑式 Dijkstra 搜索仍能取得良好加速，这也印证了这一点。对于更多源点，较大的 $k \geq 128$ 可能会有所帮助；但最终我们会遇到缓存限制，因为每个顶点都需要处理数百字节的距离标签。

单个 BCH 搜索的捆绑面临着与椭圆 BCH 搜索相同的问题。由于桶已排序，大多数桶条目扫描都发生在 BCH 查询源点附近。事实上，对于单个 BCH 查询，这个问题甚至更加明显，因为基于成本的停止准则使每次搜索的总半径更小。在 PALS 情况下，每次搜索的半径非常小，以至于即使在 $k = 8$ ，不使用 SIMD 指令进行捆绑也会增加运行时间。在 DALS 情况下，捆绑式搜索效果更好，因为停止准则较差，导致在较大距离处需要进行更多工作。借助 SIMD 并行， $k = 8$ 与 $k = 1$相比几乎没有缺点。因此，两种情况下的最优值都是 $k = 8$。

9.2 排序桶

下面，我们分析排序桶对椭圆 BCH 搜索以及单个和集体最后停靠点 BCH 搜索的影响。我们考察扫描的桶条目数量减少情况

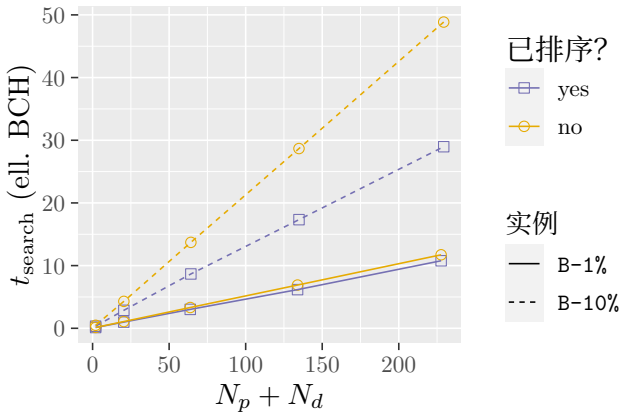


图5 排序桶对椭圆 BCH 查询运行时间的影响。在 $k = 1$ 的 `Berlin-1pct` 和 `Berlin-10pct` 实例上，展示了 $\rho \in \{$0秒、150秒、300秒、450秒、600秒$\}$时的平均运行时间（毫秒）。

以及对搜索运行时间和更新桶所需时间的影响。我们在 `Berlin-1pct` 和 `Berlin-10pct` 实例上，针对 $\rho \in \{$0秒、150秒、300秒、450秒、600秒$\}$，通过实验比较使用排序桶和未排序桶的所有搜索。对于椭圆 BCH 搜索和单个最后停靠点 BCH 搜索，我们使用 $k = 1$。对于椭圆 BCH 搜索和单个最后停靠点 BCH 搜索，使用未排序桶时我们只运行一次迭代。

椭圆 BCH 搜索的排序桶。 椭圆 BCH 搜索的桶已经通过椭圆剪枝进行了强力剪枝。因此，只有车辆数量足够多时，对这些桶进行排序才会产生显著影响。如图5所示，对于 `Berlin-1pct`实例，排序桶的影响有限；而对于 `Berlin-10pct` 实例，影响则大得多，因为后者考虑的车辆数量是前者的十倍。对于较大的输入，排序桶将扫描的条目数减少了约一半，使搜索时间最多缩短 40%（20 毫秒）。与此同时，维持桶条目的顺序只会使更新桶条目的时间增加不到 10 微秒。总之，排序桶是椭圆 BCH 搜索的一项重要改进，尤其有助于提升算法对更多车辆的可扩展性。

末站 BCH 搜索的排序桶。对于末站 BCH 搜索，排序桶对于减少扫描的桶条目数量至关重要，因为我们无法使用椭圆剪枝。图 6 展示了排序桶对末站搜索时间的影响。对于单个 BCH 搜索，在 PALS 和 DALS 两种情况下，使用排序桶扫描的桶条目分别减少了 95% 以上和 75% 以上，搜索时间最多缩短了 82% 和 78%。对于集体搜索，扫描的桶条目数量也按大致相同的比例减少。由于集体搜索扫描的桶条目总数较少，对搜索时间的影响不那么显著，两种情况下最多缩短 70%。

维护排序后的一站桶每个请求的平均开销不到 20µs，而运行时间的缩减幅度大一到三个数量级。

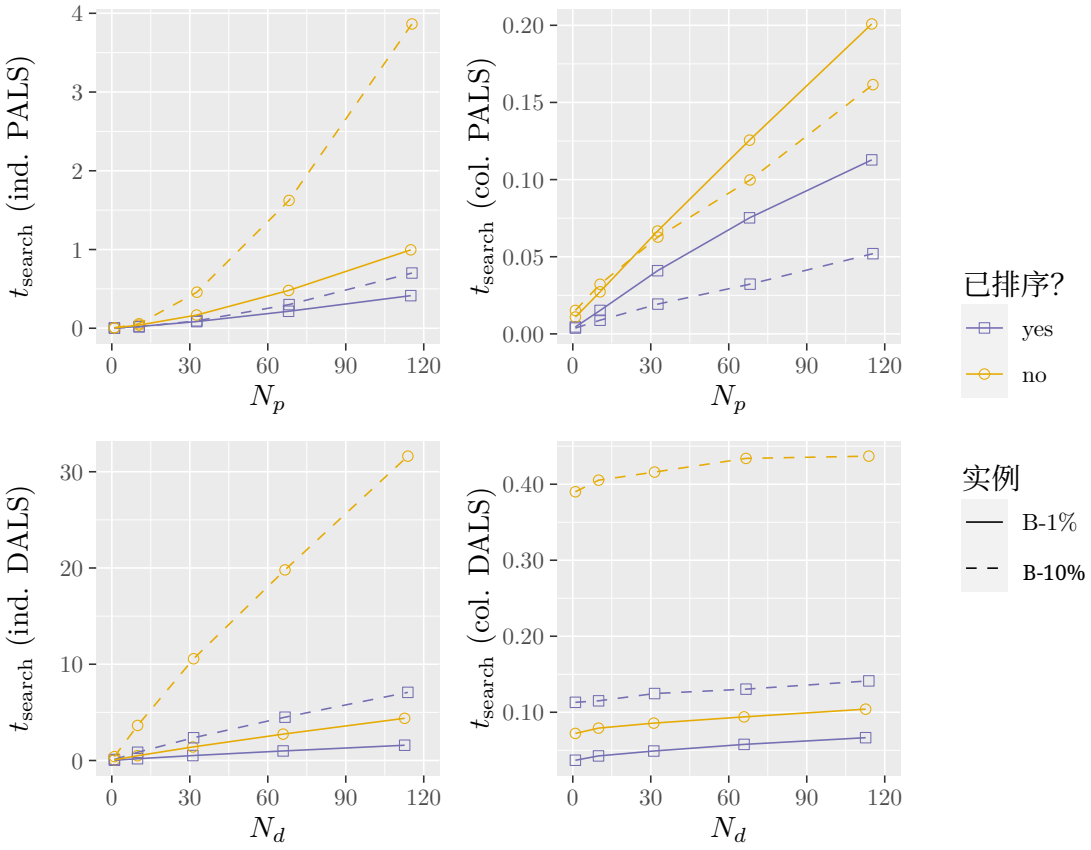


图 6 排序桶对最后一站 BCH 查询运行时间的影响。展示了在 PALS（上）和 DALS（下）情况下，针对 `Berlin-1pct` 和 `Berlin-10pct` 实例进行的单个 BCH 查询（左；$k = 1$）和集体 BCH 查询（右）的平均运行时间（毫秒），适用于 $\rho \in \{0\text{s},150\text{s},300\text{s},450\text{s},600\text{s}\}$ 有无排序桶的情况。请注意不同的 y 轴。

9.3 集体 BCH 搜索

在表 3 中，我们比较了用于 PALS 和 DALS 情况的三种搜索方法的搜索时间，以及枚举候选插入所需的时间。此外，我们还展示了松弛边和扫描的桶条目数量。我们展示了 $\rho \in \{0\text{s},300\text{s},600\text{s}\}$ 下 `Berlin-1pct` 和 `Berlin-10pct` 实例的结果。由于运行时间较长，我们只对 Dijkstra 搜索进行了一轮实验。

我们发现，在 $\rho = 0\text{s}$ 时，集体搜索比单个 BCH 搜索更慢（`B-1%` 上的 DALS 情况除外）。这是因为只有一个取件点和一个下车点，因此，与只传播距离相比，传播标签所带来的额外开销并不值得。

不过，在 $\rho = 300\text{s}$ 和 $\rho = 600\text{s}$ 时，集体搜索的搜索时间最短，枚举时间也显著优于其他方法。集体搜索的搜索时间最多比单个 BCH 搜索低一个数量级。我们认为，这归因于集体搜索的两项主要优势。

首先，由于我们使用特定 PD 对或下车点的成本下界，而非一般的成本下界，因此可以更精确地剪枝集体搜索。

表 3 在 `Berlin-1pct` 和 `Berlin-10pct` 实例上，针对 PALS 和 DALS 情况以及三个半径 $\rho \in \{0s, 300s, 600s\}$，集体 BCH 搜索（Coll.）、单个 BCH 搜索（BCH）和 Dijkstra 搜索（Dij.）的运行时间（单位：µs）比较。显示每个请求的边松弛次数（$\#_{\text{rel.}}$）、扫描的桶条目数（$\#_{\text{scans}}$）、平均搜索时间（t_{search}）以及枚举可能的插入项的平均时间（t_{enum}）。粗体标记的时间为每个半径下的最短时间。

			Berlin-1pct				Berlin-10pct			
	ρ	Alg.	$\#_{\text{rel.}}$	$\#_{\text{scans}}$	t_{search}	t_{enum}	$\#_{\text{rel.}}$	$\#_{\text{scans}}$	t_{search}	t_{enum}
PALS	0	Coll.	33	7	4.46	0.49	12	12	3.66	0.53
		BCH	30	7	**3.16**	0.37	12	7	**2.31**	0.42
		Dij.	459	–	33.26	**0.30**	179	–	14.38	**0.34**
	300	Coll.	225	34	**40.98**	**0.65**	68	33	**19.47**	**0.68**
		BCH	493	152	41.20	27.55	357	570	62.02	93.33
		Dij.	2125	–	255.21	24.64	1599	–	228.07	80.14
	600	Coll.	439	64	**112.81**	**0.79**	100	52	**52.47**	**0.89**
		BCH	2243	1135	191.32	408.40	1661	5530	420.25	1839.26
		Dij.	14821	–	1551.77	366.51	12852	–	1689.69	1681.56
DALS	0	Coll.	177	1077	36.31	3.15	154	7895	**112.76**	**6.89**
		BCH	183	1128	**26.14**	**3.12**	159	8283	116.17	7.27
		Dij.	22317	–	1978.23	28.55	15780	–	1411.81	132.11
	300	Coll.	197	1053	**48.74**	**7.46**	173	7857	**126.67**	**20.13**
		BCH	1132	4792	134.84	97.13	1035	37684	605.15	363.65
		Dij.	24762	–	3444.91	174.46	18522	–	2667.37	694.29
	600	Coll.	213	1043	**66.51**	**14.84**	189	7837	**141.38**	**40.57**
		BCH	3868	15484	432.85	754.17	3509	122199	1967.98	2403.68
		Dij.	62087	–	8149.34	1639.57	49489	–	6807.04	5992.58

每个 PD 对或下车点。这适用于桶扫描的停止条件，也适用于整个搜索过程。

其次，集体搜索会在一次搜索中考虑所有源点，从而最大限度地利用可用于支配剪枝的信息。捆绑式搜索一次只能考虑 k 个搜索，并且会产生 $k > 8$ (s.a.) 的时间开销。因此，每条边最多可能被 $N_p/8$ 或 $N_d/8$ 个捆绑式搜索扫描，且无法在不同搜索之间捆绑松弛操作。因此，对于单个 BCH 搜索，随着 PD 接送地点数量（N_p，$N_d \sim \rho^2$）增加，边松弛次数和桶条目扫描次数的增长速度远快于集体 BCH 搜索。事实上，在 DALS 情况下，支配剪枝的效果非常好，因此随着下客点数量增加，搜索时间几乎保持不变（参见图 6）。

此外，集体搜索的枚举时间也几乎保持不变，而单个 BCH 搜索的枚举时间则会随着 ρ 大幅增加。这是因为集体搜索会在搜索过程中确定单个候选插入，而单个 BCH 搜索则会先找出所有距离，然后针对候选车辆与 PD 对的每种组合分别枚举插入。由于 PD 对的数量与 ρ^4 成正比，当尝试的插入达到数万次时，枚举时间会迅速变得非常长。

9.4 与 LOUD 的比较

在本节中，我们将我们的方法与 LOUD 算法 [4]进行比较。

表4 LOUD（ρ = L-...）和 KaRRi 在不同半径（$\rho \in$ {0s、300s、600s}）下，于 `B-1%`、`B-10%`、`R-1%` 和 `R-10%` 上各阶段的运行时间（单位：µs）。展示了查找 P_ρ 和 D_ρ、PD 距离搜索、椭圆 BCH 搜索、枚举普通插入和 PBNS 插入、PALS 和 DALS 搜索、更新路径和桶的平均时间，以及每个请求的平均总时间。带星号的数字为估算值。

实例	ρ	find P_ρ, D_ρ	PD	BCH	Ord.& PBNS	PALS	DALS	更新	总计
B-1%	L-0	0	17	124	147	43	1974	84	2388
	0	2	36	123	42	3	28	132	367
	300	95	155	497	119	43	56	138	1103
	600	316	735	1703	469	156	81	140	3601
B-10%	L-0	0	16	368	415	23	1474	80	2376
	0	3	35	362	353	3	119	200	1074
	L-300	*104	*18930	28675	736	918	44071	*80	*93514
	300	104	157	1502	641	24	140	203	2772
	L-600	*353	*246521	48856	1900	8998	166244	*80	*472952
	600	353	759	4804	1741	106	181	213	8158
R-1%	L-0	0	19	173	228	128	8572	82	9202
	0	3	18	163	106	5	34	138	481
	300	98	132	668	194	62	59	146	1375
	600	293	566	1989	494	216	81	149	3807
R-10%	L-0	0	18	703	944	89	6154	80	7988
	0	3	19	708	944	5	163	298	2155
	300	106	138	3360	1293	42	182	320	5462
	600	324	595	9396	2476	128	226	387	13554

运行时间。我们在表4中给出了两种算法在 `Berlin-1pct`、`Berlin-10pct`、`Ruhr-1pct`和 `Ruhr-10pct` 实例上各个阶段的运行时间。对于 KaRRi，我们考察 $\rho \in$ {0s,300s, 600s}，并在每种配置中使用最优末站搜索方法。我们还报告了 LOUD 算法（ρ =L-0）在全部四个实例上的运行时间。此外，我们还估算了将 LOUD 朴素扩展到多个接送地点（ρ =L-300 和 ρ =L-600）后，在中等规模 `Berlin-10pct` 实例上的运行时间。

首先，我们考虑单一取件点和下车点（$\rho = 0$s）的场景，这也是 LOUD 所考虑的场景。在这种情况下，尽管扫描的桶条目数量有所减少，排序桶对椭圆BCH搜索搜索时间的影响仍然很小。我们认为，这是因为我们的实现旨在处理任意数量的接送地点，而 LOUD 专门针对 $N_p = N_d = 1$ 的情况。不过，我们的末站 BCH 搜索很适合 $\rho = 0$s。在 PALS 和 DALS 情况下，它们分别比 LOUD 的 Dijkstra 搜索快至多 27 倍和 250 倍。不过，维护排序的最后一站桶确实会增加更新时间，尤其是在桶中包含更多条目的较大实例中。总体而言，与 LOUD 相比，我们将每个请求的平均处理时间分别降低了*6.5*倍（`Berlin-1pct`）和*2.2*倍（`Berlin-10pct`）。在较大的鲁尔实例中，对于 `Ruhr-1pct` 和 `Ruhr-10pct` 实例，我们分别实现了*19.1*倍和*3.7*的加速。

接下来，我们考虑一种对 LOUD 的扩展方案的估算，该方案仅使用原始算法中的技术。为此，我们将 KaRRi 配置为不使用捆绑式搜索和排序桶，并在 PALS 和 DALS 情形下使用 Dijkstra 搜索。对于 PD 距离，

26 快速多对多路径规划：支持多次接送地点的拼车路径规划

表 5 KaRRi 在 B-1%、 B-10%、
R-1% R-10% 95 和 上使用不同半径 ($\rho \in$ {0s, 300s,600s}) 时的解的质量。对于请求，我们报告平均等待时间和 %-分位等待时间，以及平均乘车行程时间和行程时间（单位：mm:ss）。对于车辆，我们给出空驶、载客行驶和停靠所用的平均时间，以及平均总运营时间（单位：hh:mm）。

实例	ρ	wait	w.-95%	ride	trip	空	occ	stop	op
B-1%	0	3:49	9:38	12:32	17:18	0:41	3:10	0:28	4:19
	300	3:22	8:24	11:56	16:13	0:30	2:56	0:30	3:56
	600	3:27	8:34	11:47	16:11	0:29	2:51	0:30	3:51
B-10%	0	2:31	7:11	11:57	15:07	0:16	2:29	0:26	3:10
	300	2:20	6:20	11:33	14:41	0:09	2:12	0:27	2:48
	600	2:30	7:34	11:27	14:50	0:08	2:08	0:27	2:42
R-1%	0	4:49	11:48	12:25	18:26	0:56	3:15	0:27	4:39
	300	4:15	10:20	11:48	17:07	0:44	3:03	0:29	4:16
	600	4:17	9:52	11:33	16:52	0:42	2:58	0:29	4:09
R-10%	0	3:11	8:46	11:46	15:48	0:24	2:40	0:25	3:29
	300	2:45	7:07	11:13	14:51	0:16	2:26	0:26	3:08
	600	2:55	7:58	11:06	14:56	0:15	2:22	0:26	3:03

我们将 LOUD 的 PD 距离搜索时间乘以 $N_p \cdot N_d$，以此估算每个 PD 对执行一次收缩层次查询所需的时间。我们发现，捆绑和排序桶使椭圆 BCH 搜索比朴素扩展快约一个数量级。对于 PALS 和 DALS 搜索，我们的集体式桶收缩层次方法分别比标准迪杰斯特拉方法快两个和三个数量级。我们假设，实际执行 PD 距离的收缩层次查询会比估算值更快。尽管如此，值得注意的是，我们基于桶收缩层次的方法比估算值快数百倍。

解的质量。 接下来，我们初步探讨通过在拼车中加入步行，如何改善行程时间和车辆运营时间。

在表 5 中，我们比较 KaRRi 在 $\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$ 下的解的质量。请注意，如果步行时间能使成本低于拼车行程，我们便允许乘客步行前往目的地（见第4节 4.1）。此处使用的成本函数对乘客行程时间和车辆运营时间赋予相同权重（$\tau = 1$）。随着 ρ 值增大，我们观察到这两个指标均有所改善。在 $\rho = 300\text{s}$ 时，平均车辆运营时间和乘客等待时间最多改善 12%，而行程时间最多改善 7%。在 $\rho = 600\text{s}$ 时，与 $\rho = 300\text{s}$ 相比，车辆运营时间和乘客乘车时间进一步缩短。

这些初步结果中有许多参数尚未进行评估。例如，我们尚未在成本函数中考虑乘客为了减少行程时间而步行更长距离的意愿（因为 $\omega = 0$）。此外，我们仅展示了固定数量的车辆以及代表普通工作日的请求密度下的结果。未来，我们希望在分析接送地点对拼车的影响时纳入这些参数，并将此评估扩展到更大规模的输入。

10 结论与未来工作

KaRRi 利用桶式收缩层次开发高效的多对多路径规划，可在考虑多个上车和下车地点的情况下，高效调度大型车队。灵活的成本函数可适用于多种情境，例如使用步行、自行车

或滑板车。我们预计，排序桶等新技术也可应用于其他采用相关源点和目标点进行多对多路径规划的问题。

接下来，我们希望使用 KaRRi 评估不同的交通场景。例如，我们预计，与单个接送地点相比，采用速度更快的个人交通方式或使用数量更少但容量更大的车辆，能够实现更大的节省。

在算法方面，未来的工作可以通过按邻近程度将路径规划源点聚类为大小为 k 的批次、找到椭圆BCH搜索的集体方法，以及同时针对请求（使用不同车辆的请求）和不同接送地点并行化算法来实现加速。通过摒弃贪心在线调度，转而考虑预订行程，并利用机会以局部搜索式优化透明地更改现有行程，还可以进一步优化总成本。

我们预计，进一步的推广可以将 KaRRi 与公共交通相结合，使上车和下车地点可以是公交车或火车的停靠点，并且成本函数需要考虑公共交通时刻表。

从长远来看，可以允许在行程途中更换车辆。这样一来，就有可能通过使用更大型的车辆实现更多拼车，最终构建高度自适应的软件定义公共交通系统。这也带来了有趣的算法挑战，因为可能的路线选项会呈组合爆炸式增长。

参考文献

1 Reinhard Bauer 和 Daniel Delling。《SHARC：快速且稳健的单向路径规划》。《ACM实验算法杂志》（JEA），14，2010年。 doi:10.1145/1498698.1537599。2 Joschka Bischoff、Michal Maciejewski 和 Kai Nagel。《全市共享出租车：柏林模拟研究》。发表于 IEEE 第20届智能交通系统国际会议（ITSC），2017年。 doi:10.1109/ITSC.2017.8317926。3 Valentin Buchhold、Peter Sanders 和 Dorothea Wagner。《使用工程化可定制收缩层次进行实时交通分配》。《ACM实验算法杂志》（JEA），24，2019年。 doi:10.1145/3362693。4 Valentin Buchhold、Peter Sanders 和 Dorothea Wagner。《快速、精确且可扩展的动态拼车》。发表于 SIAM算法工程与实验研讨会（ALENEX），2021年。 doi:10.1137/1.9781611976472.8。5 Daniel Delling、Andrew V Goldberg、Andreas Nowatzyk 和 Renato F Werneck。《PHAST：硬件加速最短路径树》。《并行与分布式计算杂志》，73，2013年。 doi:10.1016/j.jpdc.2012.02.007。6 Daniel Delling、Andrew V Goldberg、Thomas Pajor 和 Renato F Werneck。《道路网络中的可定制路径规划》。《INFORMS交通科学》，51(2)，2017年。 doi:10.1287/trsc.2014.0579。7 Daniel Delling、Andrew V Goldberg 和 Renato F Werneck。《道路网络中更快的批量最短路径》。发表于第11届交通建模、优化与系统算法方法研讨会（ATMOS）。LIPIcs，2011年。 doi:10.4230/OASIcs.ATMOS.2011.52。8 Edsger W Dijkstra。《关于图论中两个问题的注记》。《数值数学》，1，1959年。9 Robert Geisberger、Peter Sanders、Dominik Schultes 和 Christian Vetter。《使用收缩层次进行大型道路网络中的精确路径规划》。《INFORMS交通科学》，46(3)，2012年。 doi:10.1287/trsc.1110.0401。10 Preeti Goel、Lars Kulik 和 Kotagiri Ramamohanarao。《高效拼车的最优上车点选择》。《IEEE大数据汇刊》，3(2)，2017年。 doi:10.1109/TBDATA.2016.2599936。

11 Moritz Hilger、Ekkehard Köhler、Rolf H Möhring 和 Heiko Schilling。《使用弧标记的快速点对点最短路径计算》。《最短路径问题：第9届 DIMACS 实现挑战赛》，2009年。12 Andreas Horni、Kai Nagel 和 Kay Axhausen（编）。《多智能体交通仿真 MATSim》。Ubiquity Press，2016年。 doi:10.5334/baw。13 Sebastian Knopp、Peter Sanders、Dominik Schultes、Frank Schulz 和 Dorothea Wagner。《使用高速公路层次结构计算多对多最短路径》。发表于 SIAM 算法工程与实验研讨会（ALENEX），2007年。 doi:10.1137/1.9781611972870.4。14 Catherine Morency。《拼车的矛盾性》。《Transportation》，34(2)，2007年。 doi:10.1007/s11116-006-9101-9。15 Changle Song、Julien Monteil、Jean-Luc Ygnace 和 David Rey。《拼车激励机制：社会福利与交通拥堵案例研究》。《Journal of Advanced Transportation》，2021年。doi:10.1155/2021/6627660。16 Mitja Stiglic、Niels Agatz、Martin Savelsbergh 和 Mirko Gradisar。《拼车系统中会合点的优势》。《Transportation Research Part B: Methodological》，82，2015年。doi:10.1016/j.trb.2015.07.025。17 Hiroki Yanagisawa。《全点对最短路径问题的多源标签修正算法》。发表于 IEEE 并行与分布式处理国际研讨会（IPDPS），2010年。 doi:10.1109/IPDPS.2010.5470362。18 Biying Yu、Ye Ma、Meimei Xue、Baojun Tang、Bin Wang、Jinyue Yan 和 Yi-Ming Wei。《拼车的环境效益：北京案例》。《Applied Energy》，191，2017年。doi:10.1016/j.apenergy.2017.01.052。19 Dominik Ziemke、Ihab Kaddoura 和 Kai Nagel。《MATSim 柏林开放场景：基于合成需求建模与开放数据的多模式基于智能体的交通仿真场景》。《Procedia Computer Science》，151，2019年。 doi:10.1016/j.procs.2019.04.120。