过载惩罚下的负载均衡需求分配

Sarnath Ramnath 圣克劳德州立大学，美国
sarnath@stcloudstate.edu

Venkata M. V. Gunturi 赫尔大学，英国 罗帕尔印度理工学院，旁遮普，印度
v.gunturi@hull.ac.uk

Subi Dangol 圣克劳德州立大学，美国
sdangol3@stcloudstate.edu

Abhishek Mishra 罗帕尔印度理工学院，旁遮普，印度
2018csm1002@iitrpr.ac.in

Pradeep Kumar 罗帕尔印度理工学院，旁遮普，印度 2019CSM1008@iitrpr.ac.in

arXiv:2304.06543v1 [cs.DS] 13 Apr 2023

摘要

负载均衡需求分配（LBDD）的输入由以下内容组成：(a) 一组公共服务中心（例如学校）；(b) 一组需求（人员）单元；以及 (c) 一个成本矩阵，其中包含所有需求单元—服务中心对的分配成本。此外，每个服务中心还关联一个容量概念，以及在其发生超载时产生的惩罚。给定输入后，LBDD问题确定从需求单元集合到服务中心集合的映射。其目标是确定一个映射，使以下两项之和最小：(i) 需求单元与其分配的服务中心之间的总分配成本；以及 (ii) 产生的惩罚总值。LBDD问题可应用于城市规划领域。一个LBDD问题实例可以归约为一个最小成本二部匹配问题实例。然而，这种方法无法扩展到现实世界中的大型问题实例。当前与LBDD相关的最先进方法采用了一些简化假设，例如容量无限，或总容量等于总需求。本文针对LBDD问题提出了一种新颖的基于分配子空间再调整的方法（ASRAL）。我们从理论上分析ASRAL，并给出其渐近时间复杂度。我们还在大型问题实例上对ASRAL进行了实验评估，并将其与替代方法进行比较。结果表明，ASRAL能够实现扩展，同时相较于替代方法保持显著更高的解质量。此外，我们还将ASRAL扩展为para-ASRAL，利用图形处理器和CPU 核心加速执行，同时保持与ASRAL相同的解质量。

CCS 概念

• **信息系统 →地理信息系统**。

关键词

空间网络、Voronoi 图、图算法

允许免费为个人或课堂使用而制作本作品全部或部分内容的数字副本或纸质副本，但前提是这些副本不得用于营利或商业利益，也不得为此目的进行分发，并且副本首页须保留本声明和完整引文。对于本作品中由非 ACM 所有的组件，必须尊重其版权。注明出处后可以进行摘要。若要以其他方式复制、再版、发布到服务器或分发至列表，须事先获得特别许可和/或支付费用。请向 permissions@acm.org 申请许可。会议缩写 *'XX*，*2018*年*6*月*3*日至*5*日，纽约州伍德斯托克© 2023 计算机协会。ACM ISBN 978-1-4503-XXXX-X/18/06…$ 15.00 https://doi.org/XXXXXXX.XXXXXXX

ACM 引用格式：
Sarnath Ramnath、Venkata M. V. Gunturi、Subi Dangol、Abhishek Mishra 和 Pradeep Kumar。2018。惩罚值下的负载均衡需求分配。载于论文集：请根据版权确认邮件填写正确的会议名称（会议缩写 *'XX*）。ACM，纽约，纽约州，美国，16 页。https://doi.org/XXXXXXX.XXXXXXX

1 引言

负载均衡需求分配（LBDD）问题的输入包括以下三项：(a) 一组 S 服务中心（例如新冠诊所、学校等）；(b) 一组 D 需求单元（例如人）；(c) 一个成本矩阵 CM，其中包含将需求单元 $d_i \in D$ 分配给服务中心 $s_j \in S$ 的成本（针对所有 $< d_i, s_j >$ 对）。此外，每个服务中心 $s_j \in S$ 都关联着一个正整数容量，以及一种“惩罚”，表示超载该服务中心时必须支付的“额外成本”。给定这些输入，LBDD问题要确定需求单元集合与服务中心集合之间的映射。其目标是确定一种映射，使以下两项之和最小：(1) 分配总成本（所有分配的累计成本）；(2) 服务中心超载时产生的惩罚总额（如有）。

问题动机：负载均衡需求分配（LBDD）问题可应用于城市规划领域。例如，可以考虑这样一项任务：划定服务中心的运营地理区域，例如学校、新冠肺炎诊所和免预约新冠病毒检测中心（用于在城市范围内持续监测疾病）。

这里的关键在于，前述各类服务中心都具有一般意义上的容量概念。容量决定了每天、每周或在任何特定时段内（例如患者患病的典型时长）能够（舒适地）接待的人数（需求）。此外，如果分配给某个服务中心的人数显著超过其容量，其服务质量预计会下降。在这些约束条件下，确定需求与服务中心之间的“最优”映射，可以归结为我们的LBDD问题。下面以免预约新冠病毒检测中心为例，说明一个LBDD问题实例。同样，也可以轻松地将学校作为服务中心来定义一个LBDD实例。

考虑这样一个任务：为城市中的每个免预约新冠病毒检测中心划定其服务区域。可以想见，任何检测中心在检测试剂盒、实验室技术人员及其他相关设备方面的资源都有限。这些

基础设施限制决定了其容量，即每天能够检测的人数。如果到这家诊所的人数明显超过其容量，整体服务时间可能会受到影响。这一因素可通过惩罚来建模。将某人分配给特定服务中心的成本，可以根据最短距离、出行时间或沿最短路径出行的成本来定义。在这些约束条件下确定服务区域的问题，可以转化为我们的 LBDD 问题。

挑战： 从理论上讲，一个 LBDD问题实例可以归约为一个最小费用二分图匹配问题的实例 [3]。然而，最小费用二分图匹配的最优算法无法扩展到大型 LBDD问题实例。例如，我们的实验发现，对于一个仅包含 13 个服务中心和约 4000 个需求顶点的 LBDD实例，最小费用二分图匹配的最优算法需要约 7 小时才能运行完成，并占用约 40 *GB RAM*。由此不难预料，最小费用二分图匹配的最优算法无法扩展到大型城市规模问题实例。

相关工作的局限性 与我们的工作最相关的当前先进研究包括无容量网络沃罗诺伊图领域的研究 [7, 13],容量约束下的网络沃罗诺伊图 [14, 15, 18, 19],加权Voronoi图 [1] 以及最优位置查询（例如， [8, 16, 20]）。

在不考虑容量的网络沃罗诺伊图领域中， [7, 13] 假设服务中心具有无限容量，这一假设不适用于许多现实场景。另一方面，考虑容量的网络沃罗诺伊图领域中的研究 [14, 15, 18, 19] 并未考虑“超载惩罚”这一概念。只要存在有可用容量的服务中心，它们就会以迭代方式进行（需求单元的）分配。换言之，当所有服务中心都已满载（就其容量而言）时，分配便会停止。本文在更一般的设定下考虑这一问题，即允许分配超出服务中心的容量。在服务中心满载后，我们使用超载惩罚这一概念来指导后续分配。

加权Voronoi图 [1] 是专门的Voronoi图。在这些图中，将需求单元 x 分配给服务中心 p 的成本是以下两项的线性函数：(i) x 与 p 之间的距离，以及 (ii) 一个实数，表示 p 的权重，即 $w(p)$。LBDD问题不同于加权Voronoi图。与加权Voronoi图不同，我们的“$w(p)$”是服务中心 p已获得分配数量的函数。只有当分配数量超过容量时，它才会返回非零值。而在 [1]中，假设 $w(p)$始终发挥作用。

最优位置查询（例如 [8, 20]）领域的研究工作侧重于确定一个新位置，以启动新的设施，同时优化某个目标函数（例如客户与设施之间的总距离）。而在 LBDD中，我们已经有一组正在运行的设施，并且希望对其周围的需求进行负载均衡。

我们的贡献： 本文做出了以下贡献：

(a) 我们为 LBDD问题提出了分配子空间再调整这一概念。

(b) 基于分配子空间再调整这一概念，我们提出了一种用于求解 LBDD问题的新型 *ASRAL* 算法 。(c) 我们通过理论分析和实验（使用大型问题实例）对 *ASRAL* 进行了评估。

(d) 我们的实验结果表明，ASRAL 能够扩展，同时保持比其他方法显著更好的解质量。

(e) 我们还扩展了 ASRAL 算法，开发了一种并行算法，以利用日益普及的多核和多处理器系统。我们将 ASRAL 的并行实现称为 *para-ASRAL*，它同时利用图形处理器和 CPU 核心来缩短执行时间，同时保持与 ASRAL 相同的解质量。

(f)我们的理论分析表明，当服务中心受到容量限制，即不允许超载时，ASRAL 可以轻松推广并获得最优结果。

概要： 本文其余部分安排如下：第 2 节介绍基本概念和问题陈述。第 3 节介绍我们提出的方法。第 3.6 节介绍我们对 ASRAL 算法进行并行实现（para-ASRAL 算法）的构想。第 4 节对所提方法进行详细的理论分析并分析其复杂度。第 5 节对我们的方法及相关工作进行实验评估。第 6 节总结全文。附录 B 介绍了 ASRAL 算法的一种简单推广，该推广可在不允许服务中心超载的情况下计算最优解。

符号	含义
D	输入中给定的 n 需求单元集合
S	输入中给定的 k 服务中心集合
$q_{s_i}0$	服务中心的惩罚函数 s_i
A，B，F	我们用花体字母表示分配集合。分配包含以下对：$< d_i, s_j >$。其中，需求单元 d_i被分配给服务中心s_j
CM(d_i， s_j)	分配的成本（正整数）：需求单元 d_i 至服务中心s_j
$\Gamma(\mathcal{A})$	分配子空间图，表示该分配的A。此图的节点为服务中心，边表示转移服务中心之间的需求单元。

表 1：论文中使用的符号

2 基本概念与问题定义

定义 1. 服务中心 是某种特定类型的公共服务单位（例如，城市中的学校、医院、新冠病毒检测中心）。服务中心集合表示为 $S = \{s_1, ..., s_k\}$.

定义 2。 一个需求单元 表示一个有意访问先前定义的服务中心的单元群体。需求单元集合表示为 $D = \{d_1, ..., d_n\}$， n 为需求单元的数量。

定义 3。 服务中心的容量 (c_{s_i}) 是规定的需求单元数量，即服务中心 s_i能够轻松容纳的数量。例如，新冠肺炎检测设施一天（或一周）能够检测的人数，即可界定其容量。同样，学校能够招收的学生人数也符合本文所定义的容量概念。

定义4. 服务中心的惩罚函数 ($q_{s_i}()$) 是一个函数，用于返回每次向已耗尽容量的服务中心 s_i新增分配时必须支付的"额外成本"（> 0） c_{s_i}。如果分配未超出服务中心的容量，则无需支付惩罚。现实中的惩罚成本示例包括为学校增建基础设施和/或增聘教职员工的成本。

惩罚函数会考虑 s_i的当前状态（即已向 s_i添加了多少个节点），然后返回超出其容量的 j^{th} （$1 \leq j \leq (n - c_{s_i})$）分配所对应的惩罚值。 $q_{s_i}()$仅返回正的非零值，并且在 j （$1 \leq j \leq (n - c_{s_i})$）上单调递增。其直观含义是，随着超载程度持续增加，可能需要为学校（或诊所）增添越来越多的资源。

定义 5。 需求-服务成本矩阵 CM： 包含将需求单元 $d_i \in D$ 分配给服务中心 $s_j \in S$ 的成本（正整数）（对于所有 $< d_i, s_j >$ 对）。分配成本可以表示诸如沿道路网络的最短距离、出行时间和/或沿最短路径从 d_i 到s_j 的旅行成本（以美元计）、大地测量距离或欧几里得距离等。

定义 6. 分配： 分配是一个包含成对元素 $< d_i, s_j >$ 的集合。在这里，需求单元 d_i被分配给服务中心s_j。我们使用数学字母（例如，$\mathcal{A}$、$\mathcal{B}$）来表示分配。

定义 7。 完整分配： 当且仅当一个分配的基数为 n （即需求节点的数量）时，称其为完整分配。

2.1 问题陈述

现在，我们通过详细说明输入、输出和目标函数，正式定义负载均衡需求分配问题：**给定**：

- 一组服务中心 $S = \{s_1, \ldots\ldots s_k\}$。
- 一组需求单元 $D = \{d_1, \ldots\ldots d_n\}$。
- 一个需求-服务成本矩阵 CM，其中包含将需求单元 $d_i \in D$ 分配 $s_j \in S$ 至服务中心 $\forall d_i \in D(, \ \forall s_j \in S)$ 的成本。
- 每个服务中心 c_{s_i} 的容量 ($s_i \in S$)。
- 每个服务中心 q_{s_i} 的惩罚函数 ($s_i \in S$())。

输出：使以下目标函数最小化的完整分配。

目标函数：

$$\textit{Minimize}\left\{ \sum_{\substack{s_i \in Service \\ Centers}} \left\{ \sum_{\substack{d_j \in Demand\ unit \\ allotted\ to\ s_i}} CM(d_j, s_i) \right\} + Total\ Penalty\ across\ all\ s_i \right\} \quad (1)$$

2.2 在现实生活中实例化 LBDD问题

为了在现实生活中实例化 LBDD问题，我们需要定义分配成本（以 CM 为单位）和惩罚值。如前所述，分配成本可以用最短距离、沿最短路径行进的成本、大地测量距离等来定义。惩罚值可以按照成本（以美元计）来定义，例如为服务中心增加基础设施所需的成本，或因超载造成的额外响应延迟。请注意，为了使结果易于解释，分配成本和惩罚值应采用共同的"计量单位"。在处理新冠诊所、新冠病毒检测中心、学校等公共服务单位时，成本（以美元计）可以作为这样的通用计量单位。附录D提供了另一个以学校为服务中心的 LBDD问题实例。

3 所提方法

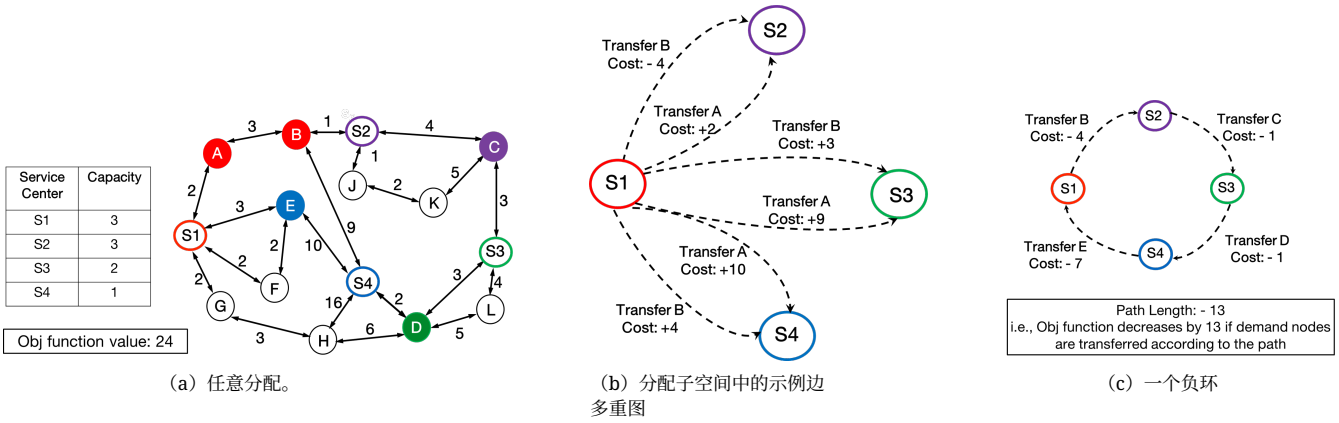
本节介绍了我们针对 LBDD问题 所提出的基于分配子空间再调整的方法（ASRAL）。本节其余内容安排如下。第 3.1 节概述所提方法。第 3.2 节介绍分配子空间再调整这一核心计算思想。第 3.3 节和第 3.4 节描述了将在后文的 ASRAL 算法中使用的一些关键子程序。第 3.5 节详细介绍所提 ASRAL 算法。第 3.6 节介绍 ASRAL 的并行实现（称为 para-ASRAL）。

3.1 算法概述

总体而言，我们的算法采用增量策略来构建解。首先，我们为所有需求单元 $d_i \in D$确定"最适合"的服务中心（使用输入中给定的 CM）。需求单元 d_i的"最适合"服务中心定义为服务中心 s_{min}，在 CM 中， d_i 与该中心之间的分配成本最低。接着，我们将元组 <需求单元 d_i及其对应的 s_{min} > 插入堆中。该堆按照元组的分配成本（相对于各自的 s_{min}）排序。在每次迭代中，使用取最小值操作从堆中选出下一个要处理的需求单元。设（任意一次迭代中）取最小值操作的结果为需求单元 d_i，其"最适合"的服务中心为 s_{min}。现在有两种情况：$s_{\min}$ **剩余容量（简单情况）：** d_i被分配给 s_{min}，目标函数增加"CM(d_i, s_{min})"这一数值。

$s_{\min}$ **已满：** 这里我们也将 d_i分配给 s_{min}。然而，这种分配会导致惩罚。因此，目标函数增加了一个值"CM(d_i, s_{min}) + 在 s_{min}处的惩罚"

现在，我们尝试调整当前分配，以降低目标函数值，并尽可能抵消惩罚的影响。这些调整分为以下两类。首先，我们进行不会导致总惩罚发生任何变化的调整。具体而言，就是移除所有负环，这一过程将在下文称为分配子空间多重图中进行（详情见下一节）。随后，我们尝试通过另一系列调整，将 s_{min}上产生的超载"转移"到其他服务中心。这是通过从分配子空间多重图中移除最负路径（起点为 s_{min}）来实现的。注意



（a）任意分配。 （b）分配子空间中的示例边多重图 （c）一个负环

图 1：示例分配中分配子空间多重图的示意图。顶点 S1、S2、S3 和 S4 均为服务中心。分配给某个服务中心的需求单元用与该服务中心相同的颜色着色，例如，在子图 (a) 中，需求单元 A 分配给了 S1。

为了确定最负路径，分配子空间多重图中不得存在任何负环。我们的算法保证了这一点：无论是在第一组重新调整之后（即移除负环），还是在将超载转移到最佳可能位置之后（即移除负路径），均满足这一条件。

3.2 分配子空间调整

本文的关键之处在于，我们所称的分配子空间多重图中的调整这一概念。给定问题实例中的每个服务中心，在该分配子空间多重图中构成一个节点。多重图中的每条边表示一个需求单元从一个服务中心转移到另一个服务中心的潜在转移。形式化定义如下。

定义 8. 给定由形如 $< d_i, s_j >$（需求单元 d_i 分配给 s_j）的元组组成的分配 $\mathcal{A}$，分配子空间多重图 $\Gamma(\mathcal{A}) = (M, N)$ 定义如下：

- 节点集合 (M)：每个服务中心 $s_i \in S$ 都是 M 中的一个节点。
- 边集合 (N)：每条边都定义为三元组 $e = (s_i, s_j, dn)$。这里，e 表示需求单元 dn 的潜在转移（该单元当前分配给 $\mathcal{A}$ 中的 s_i），从 s_i 转移到 s_j。e 的成本定义为 $\mathcal{CM}(dn, s_j) - \mathcal{CM}(dn, s_i)$。此成本称为需求单元 dn 的转移成本。

请看图1(a)，它展示了一个示例问题实例的任意分配。请注意，图1(a)仅展示任意分配（而非按照某个流程进行分配），以便于说明负环的概念。为便于理解，该图展示了一个 LBDD 实例，其中服务中心和需求单元均位于一个以有向图表示的道路网络中。这里，节点 $S1$、$S2$、$S3$ 和 $S4$ 是服务中心，而其他节点（例如 A、B、C 等）是需求单元（人）。在此问题中，将需求单元 v 分配给服务中心 r 的成本，是 v 与 r 之间的最短距离。例如，将需求单元 B 分配给 $S1$ 的成本，是图中 B 与 $S1$ 之间的最短距离（本例中为 5）。图中还标出了每个服务中心的总容量。在分配

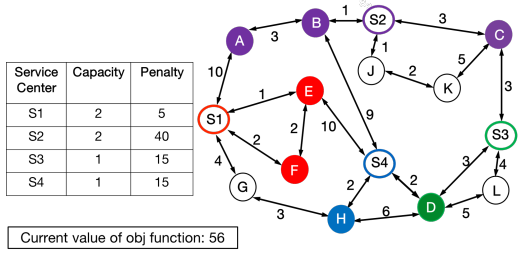
如图1(a)所示，分配给某个服务中心的需求单元会填充为与该服务中心相同的颜色。例如，需求单元 A、B 被分配给资源单元 $S1$。尚未分配的节点则不作填充。

此分配的分配子空间图将包含四个节点（每个服务中心各一个）。任意两个服务中心 s_i 和 s_j 之间都会有表示需求单元在服务中心之间转移的有向边。指向 s_j（从 s_i 出发）的边表示需求单元转移至 s_j（从 s_i 转出）。

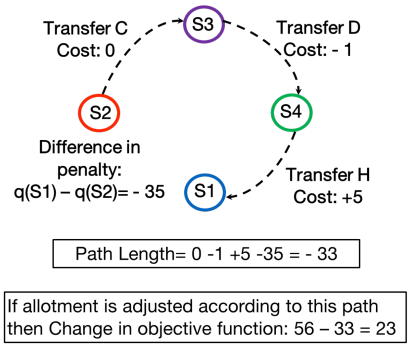
图1(b)展示了图1(a)所示分配的分配子空间图($\Gamma()$) 中的一些边。为保持清晰，我们不展示图中的所有边。图1(b)仅展示尾节点为服务中心 S1 的边。现在，鉴于根据图1(a)，S1被分配了两个需求节点(A an B)，分配子空间图中与 S1 对应的节点将有六条出边（分别有两条边指向其他各个服务中心）。例如，考虑图1(b)中从S1指向 S2 的两条边。其中一条表示需求单元 A（转移到S2），另一条表示将需求单元 B 转移到服务中心 S2。边的成本定义为服务中心的分配成本之差。例如，图1(b)中对应于 A 的边的成本定义为$\text{CM}(A, S2) - \text{CM}(A, S1)$（在我们的示例中为 2）。请注意，在某些情况下，分配子空间图中的边成本可能为负。

3.2.1 使用分配子空间多重图改进解。分配子空间多重图有助于通过以下两种方式改进当前解：(a) 负环消除；(b) 负路径消除。需要注意的是，这两种方式并非完全相互独立。在消除负路径之前，我们必须先确保分配子空间多重图中不存在负环（我们的算法会确保这一点）。下面我们将在概念层面讨论这些思路。有关实现的详细信息，请参阅第 3.3 节和第 3.4 节。

通过负环消除进行调整：考虑图1(c)所示的环。这个有向环出现在图1(a)所示分配的分配子空间多重图中。该环的总成本为 −13。换句话说，如果我们通过转移该环中各边对应的需求单元来调整分配（如图1(a)所示）（例如，将 B 转移到 S2，将 C 转移到 S3，将 D 转移到 S4，将 E 转移到 S1），那么目标函数值将减少 13 个单位。



(a) 任意分配。



(b) 分配子空间图中的负路径

图2：展示分配子空间多重图中的负路径。顶点 S1、S2、S3 和 S4 是服务中心。分配给服务中心的需求节点以与该服务中心相同的颜色着色。

通过负路径消除进行调整：考虑图2(a)所示的任意分配。与前一种情况类似，我们采用了任意分配（而非遵循某个过程），以便更直观地说明负路径的概念。此外，为便于理解，图2(a)所示的问题实例假设惩罚函数是整数，而非定义4中所述的单调函数。

图2(b)展示了图2(a)所示分配对应的分配子空间多重图中的一条路径。在这条路径上，我们将需求单元 C 转移到 S3，将 D 转移到 S4，最后将 H 转移到 S1。另外请注意，S2 最初处于超载状态（如图2(a)所示）。因此，如果我们按照上述路径调整分配，那么 S2 支付的总惩罚将会减少，因为它正在将一个需求单元（C）转移到另一个服务中心。另一方面，由于 S1 接收了超出其容量的需求单元，我们现在必须支付 S1 的惩罚。以下是该路径总成本的表达式：

路径成本 = 转移成本（C 到 S3） + 转移成本（D 到 S4） +转移成本（H 到 S1） + S1 的惩罚 - S2 的惩罚= **-33**

因此，如果我们按照图2(b)所示的路径调整初始分配（如图2(a)所示），那么目标函数值将减少 33。请注意，在我们的实现中，每次迭代后都不会重新创建 Γ()。相反，我们通过对多重图进行增量修改来维护它。更多细节将在后文介绍。

3.3 移除负环的过程

任意分配 A 的分配子空间多重图 (Γ(A)) 都可能包含多个负环。移除所有负环的计算开销会很大。为此，在实际实现这一概念时，我们采用增量方法，将需求单元逐个加入分配中。每加入一个需求单元后，我们都会确定并移除可能因这次添加而产生的负环。

更具体地说，设 A 为当前分配，现将一个尚未处理的需求单元 d_i 分配给服务中心 s_j。令所得分配为 A′。由于对 Γ(A) 的唯一修改是针对从 s_j 出发的边，如果 Γ(A) 不含负环，那么 Γ(A′) 中的任何负环都必须经过 s_j。在引理1中，我们将证明，如果移除 Γ(A′) 中最负的环，那么所得分配 A′′ 将满足 Γ(A′′) 中不含负环。为了计算最负的环，我们确定“指定的”服务中心（在我们的示例中为 s_j），并创建一种分配子空间多重图的变体，称为 负环分配子空间图。这些概念在定义 9 中予以形式化。

定义 9。令未处理的需求单元 d_i 被添加到服务中心 $s*$。令所得分配为 $\mathcal{A}$，其分配子空间多重图为 $\Gamma(\mathcal{A})$。负环分配子空间图，$NCyc(\Gamma(\mathcal{A})\ s*) = (P, Q)$，定义如下：

(1)节点集合*(P)*：每个服务中心 $s_l \in S - s*$ 是 P中的一个节点 。服务中心 $s*$ （称为锚点或特殊服务中心）由两个节点 $s*^{in}$ 和 $s*^{out}$表示。

(2) 边集合*(Q)*: 任意两个节点之间的边的成本 $u, v \in P$ 定义如下：*(a)*如果 $u \notin \{s*^{in}, s*^{out}\}$ 且 $v \notin \{s*^{in}, s*^{out}\}$，则 $cost(u, v)$ 是从 u 到 v 在 $\Gamma(\mathcal{A})$ 中有向的所有边的成本中的最小值（出现相同值时任意打破平局）。

(b) 如果 $u = s*^{out}$，那么 $cost\ (u, v)$ 是所有从 $s*$ 到 v 且位于 Γ（A）中的边的成本最小值（如有并列，任意打破平局）。

(c) 如果 $v = s*^{in}$，那么 $cost\ (u, v)$ 是所有从 u 到 $s*$ 且位于 Γ（A）中的边的成本最小值（如有并列，任意打破平局）。

(d) 所有其他边的成本均为 0 。

创建负环分配子空间图（$NCyc\Gamma()$）后，我们确定锚定服务中心的“in”和“out”副本（s_j^{in}和 s_j^{out}）之间的最低成本路径。请注意，由于 s_j^{in}和 s_j^{out}都映射到s_j 中的 Γ(A)，因此该路径给出了经过s_j 在 Γ(A)中的最低成本环。为计算最低成本路径，我们使用 Bellman-Ford 标号修正算法。

根据该最低成本路径的边调整当前分配 A，可以消除负环。请注意，我们不能采用标号设定方法（例如 Dijkstra算法）来寻找 $NCyc\Gamma()$中的最低成本路径，因为其中可能包含负边。该过程的详细伪代码见算法1。

算法1负环细化

输入： (a) 当前分配 A；(b) 锚定服务中心 SC_θ
输出 调整后的分配 B，使得其中存在
其分配子空间中不存在负环。

```
1: 创建 the 负环 分配 子空间 图 NCycΓ(A SC_θ)，以 SC_θ作为锚点顶点。
2: 令 w_out和 w_in分别为锚点顶点的"IN"和"OUT"副本 SC_θ 在 minCycΓ(A)中。
3: 在 NCycΓ(A SC_θ) 中查找从 w_out到的最低成本路径 w_in
如果 P_min < 0 的成本则
    对于 每个需求节点转移 < s_old， dn， s_new > ∈ P_min 执行
        B ← 通过从 s_p 中移除 dn 并将其添加到，来更改 A s_q.
        更新 ΓDS
        A ←B
        Δ ← Δ + 对应边的成本，位于 P_min /* CM(dn, s_p) − CM(dn, s_q) */
    结束循环。
结束 if
12: 返回新的分配 B
```

3.4 移除负路径过程

考虑这样一种情况：在主分配算法的某次迭代中，将未处理的需求单元 d_i 分配给服务中心 s_j，且该分配导致s_j处产生惩罚。令所得分配为 B。如前所述，相应的 Γ(B)可能存在一条经过s_j 的负环。首先，使用上一节所述的过程，从 Γ(B)中移除该负环。令所得分配为 B$\prime$。现在，我们尝试通过寻找一条从 Γ(B$\prime$)中的 s_j 出发的最负路径来降低目标函数的值。具体过程将通过下文定义的 *NegPath* 分配子空间图 完成。

定义 10。 将未处理的需求节点 d_i 分配给服务中心 $s*$。令 $\mathcal{B}3$ 表示从 Γ($\mathcal{B}$)中移除负环后所得的分配。*NegPath* 分配子空间图$NPath\Gamma(\mathcal{B}3, s*) = (R, S)$ 定义如下：

(1) 节点集合 *(R):* 每个服务中心 $s_i \in S - s*$ 都是 P中的一个节点。服务中心 $s*$ （称为锚定服务中心）由两个节点 $s*_{in}$和 $s*_{out}$表示。

(2) 边集 *(S)*：任意两个节点 u 、 $v \in Q$ 之间的边的成本定义如下：

(a) 如果 $u \notin \{s*_{in}, s*_{out}\}$和 $v \notin \{s*_{in}, s*_{out}\}$，那么 $cost(u, v)$是 Γ($\mathcal{B}\prime$) 中所有从 u 指向 v 的边的成本的最小值（若有多个则任意选择）。

(b) 若 $u = s*_{out}$，则 $cost(u, v)$是所有从 $s*$ 指向 v 、位于 Γ($\mathcal{B}3$) 中的边的成本最小值（若有并列，则任意选择）。

(c) 如果 $v = s*_{in}$，则 $cost(u, v) = u$ 处的惩罚（$q_u()$）- $s*$ 处的惩罚（$q_{s*}()$）。 *(d)* 如果 u 和 v 相同，则 $cost(u, v) = 0$，即自环的0成本。

创建 NegPath 分配子空间图（$NPath\Gamma()$）后，我们确定锚定服务中心的"入"和"出"副本（$s*_{in}$和 $s*_{out}$）之间的最低成本路径。为了计算最低成本路径，我们使用 Bellman-Ford 算法的标号修正算法，因为 $NPath\Gamma()$ 可能包含权重为负的边。根据这条最低成本路径上的边调整当前分配（设为 B3），即可消除这条负路径。移除最负路径后，我们就可以添加下一个需求单元。不过，我们需要确保移除负路径不会引入任何负环。在引理2中，我们证明，如果从 Γ(B3) 中移除最负路径，那么得到的分配 B4 将不会包含负环 Γ(B4。) 该过程的详细伪代码见算法4（附录A）。它与负环细化的伪代码非常相似。

3.5 ASRAL 算法的详细说明

本节详细介绍如何通过负环和负路径实现前文讨论的分配子空间再调整。我们的算法使用以下数据结构。

需求-服务成本矩阵CM的构建：根据底层问题空间的性质，可以通过以下方式确定 CM。如果服务中心和需求单元都嵌入在道路网络中，可以使用 Floyd-Warshall 算法 [5] 确定 CM。也可以为此使用其他最短路径算法（例如 [2, 6, 9–11]）。另一方面，如果服务中心和需求单元分布在连续的地理空间中，则可以使用欧几里得距离或大地测量距离计算 CM。

分配子空间多重图实现（ΓDS）在我们的实现中， Γ(A) 通过使用 $2\times\binom{k}{2}$ 个最小堆来维护，其中 k 是服务中心的数量。基本上，每一对有序服务中心对应一个堆。该堆称为 *BestTransHeap*，按照需求单元（在这对服务中心之间）的转移成本排序。对于任意一对有序服务中心 $<s_i, s_j>$，其对应的 $BestTransHeap_{ij}$ 堆顶保存着转移成本最低的需求单元（当前分配给 s_i），该单元转移到 s_j 的成本最低。在算法运行期间，每当有需求单元从服务中心 s_i 添加或删除时，与 s_i 对应的 $k-1$ 个堆都会相应更新。

$NCyc$(A $s*$) 和 $NPath\Gamma$(A $s*$) 均可由 Γ(A) 构造。对于 $NCyc\Gamma$(A $s*$)，服务中心 s_i与s_j之间的边本质上就是 $BestTransHeap_{ij}$堆的堆顶。$NPath\Gamma$(A $s*$)中的边也以类似方式构造，但惩罚差异边除外（参见定义 10），此类边可在 $O(1)$ 时间内添加。在我们的实现中， $NCyc$(A $s*$) 和$NPath\Gamma$(A $s*$) 均只创建一次，之后随着每个需求单元的加入、负环和负路径的出现而持续维护

改进。

算法2给出了我们提出的用于 LBDD 的基于分配子空间重调整的方法（*ASRAL*）算法的伪代码。我们从使用 CM 为每个需求单元 d_i确定“最合适”的服务中心开始。需求单元 d_i 的“最合适”服务中心 s_{min} 定义为对 d_i具有最低分配成本的服务中心。随后，我们将元组 <需求单元 d_i及其对应的 s_{min} >插入堆中（称为最小距离堆）。该堆按照元组对应服务中心 s_{min}的分配成本排序。每次迭代中，使用取最小值操作从最小距离堆中选取下一个待处理的需求单元。

设取最小值操作的结果为需求单元 d_i，其“最合适的”服务中心为 s_j。 d_i被分配给s_j。随后，按照以下两种情况之一更新由此得到的分配（记为 A2）：(a) s_j 有剩余容量可用于分配; 或 (b)s_j 已满。

情况*(a)*：我们将目标函数 (Δ) 增加 CM(d_i, s_j)。随后，我们移除可能由最近一次分配产生的任何负环（从Γ(A2) 中），使用算法1完成此操作。

情况*(b)*：我们将目标函数 (Δ) 增加 CM(d_i, s_j) + q_{s_i}()。其中， q_{s_i}() 是服务中心 s_i 的惩罚函数。随后，我们首先移除可能由最近一次分配产生的任何负环（使用算法1）。设 A3 为所得的分配。之后，我们移除从 s_i 出发的最负路径（在算法中称为锚点顶点），该路径位于 Γ(A3) 中。为此使用算法4。

算法2 基于分配子空间重新调整的 LBDD 方法
用于 LBDD

输入： (a) 需求-服务成本矩阵 CM；(b) 一组服务中心 S。对于每个服务中心 s_i，其容量为 c_{s_i}，并且惩罚函数 q_{s_i}()；(c) 一组需求顶点 D;
输出需求单元到服务中心的映射。

```
1: 对于每个 d_i ∈ D，确定“最合适”的服务中心
   并使用所有 <需求单元最优项创建一个最小距离堆
   服务中心> 对
2: 初始化目标函数 Δ ← 0
3: 初始化当前分配 i = 0; A1 ← NULL; 初始化 ΓDS
   /*前置条件: Γ(A1) 没有负成本环*/
4: 当最小距离堆不为空时执行
5: i++; < d_i, s_j > ← 从最小距离堆中提取最小值
6:    A2 ← 通过将 d_i 分配给s_j来更改 A1。更新 ΓDS
7:    A3 ← 负环细化（A2, s_j）（使用算法 1）
   /*不变量: A3没有负成本环*/
8:    如果 s_j 有空缺则
9:       减少s_j的容量
10:      Δ ← Δ + CM(d_i, s_j) /*增加目标函数 */
11:      A1 ← A3
12:   否则/*分配导致s_j 处的惩罚*/
13:      Δ ← Δ + CM(d_i, s_j) + q_{s_j}()
14:      A4 ← 负路径细化(A3,s_j)（使用算法 4）
   /*不变量: Γ（A4） 不存在负成本环*/
15:      A1 ← A4
16:   结束 if
17: 结束循环
```

3.6 Para-ASRAL 算法

本节介绍我们提出的 Para-ASRAL 算法，该算法利用图形处理器和 CPU 核心加速执行。我们通过以下方式实现加速：（a）在负环细化和负路径细化过程中使用并行 Bellman-Ford 算法；（b）在子空间重新调整期间使用线程更新 ΓDS。下面详细介绍这些技术。

3.6.1 并行 ***Bellman-Ford*** 算法。 负环细化和负路径细化过程都使用 Bellman-Ford 算法，分别计算 P_{min} 在 $NCyc\Gamma()$ 和 $NPath\Gamma()$ 中的最低成本路径。然而，鉴于该算法在我们的问题中的复杂度，这一过程耗时较长。更准确地说，当我们在 ($O(|E||V|)$ 中设置 $|E| = k^2$ 和 $|v| = k$ 时（Bellman-Ford 算法的时间复杂度），其复杂度在我们的问题中为 $O(k^3)$。

我们实现了 Bellman-Ford 算法的并行版本，使用图形处理器对 Bellman 算法中遍历所有边的内层循环进行并行化。我们使用 JCuda [17] 实现了该并行版本。JCuda 是 CUDA 运行时库的 Java 绑定，可让我们在图形处理器上运行 Java 代码。该过程的详细伪代码见算法3。

并行 Bellman-Ford 算法的输入如下：1） $NCyc\Gamma()$（或 $NPath\Gamma()$）的邻接矩阵表示，2）距离数组，3）父节点数组，以及 4）一个源节点（锚点顶点）。我们在 CPU 上为输入数据结构分配空间主内存（称为主机内存）并对其进行初始化。使用 JCuda，我们还会在 GPU 内存（称为设备内存）中为数据结构分配空间。包含数据结构初始值的数据将从主机内存转移到设备内存。计算完成后，最终结果将转移回主机内存。

在 Bellman-Ford 算法中，th 迭代的输出依赖于 i^{th} 迭代。但是，每次迭代中对边的松弛操作（即 Bellman 算法的内层循环）彼此独立。这使我们能够利用图形处理器的核心。我们使用 JCuda 在图形处理器上运行所有边的松弛操作（即 Bellman 算法的内层循环部分）。由于外层循环的各次迭代之间存在依赖关系，因此我们需要在每次迭代后同步图形处理器，否则将无法获得准确的结果。

所有迭代结束后（即算法 3 第 5 行的 for 循环结束后），我们将数据从设备内存转移到主机内存，并释放所有设备内存。

3.6.2 并行更新 ΓDS 并行进行。 在负环细化（算法 1 第 3 行）和负路径细化过程（算法 4 第 3 行）中找到最负路径 P_{min} 后，ASRAL 算法会根据 P_{min} 路径调整分配和 ΓDS。每条边 $e \in P_{min}$ 表示为一个元组 $< s_{old}, dn, s_{new} >$，其中我们将一个需求单元 dn 从服务中心 s_{old}转移到 s_{new}。此过程涉及更新分配和 ΓDS。

根据 P_{min} 更新分配，涉及按照边 <在服务中心之间移动需求单元

算法 3 用于寻找负路径的并行 Bellman-Ford 算法

输入：(a) $NCyc\Gamma()$（或$NPath\Gamma()$）的邻接矩阵表示；(b) 大小为 k 的距离数组；(c) 大小为k的父节点数组；(d) 一个源顶点（锚点顶点）

输出：返回距离数组和父节点数组中的结果。距离数组包含从源节点到给定图中所有其他节点的最短路径距离。父节点数组包含从源节点出发的最短路径中的父节点。

1：初始化距离数组。除源节点外，将所有节点的值设置为 ∞，并将源节点的值设置为 0。2：初始化父节点数组。将所有节点的值设置为 -1。/*两个数组均位于中央处理器，即主机内存中*/ 3：使用 JCuda 在图形处理器上为三个数组（$NCyc\Gamma()$ 或 $NPath\Gamma()$ 的矩阵表示、距离数组和父节点数组）分配内存，大小与主机内存相同。/*即设备内存*/ 4：使用 JCuda 将主机内存中的数据转移到设备内存（即图形处理器）5：**对**图中的每个节点**执行**6：使用图形处理器松弛所有边（Bellman-Ford 算法的内层循环）7：同步图形处理器 /*等待完成*/ 8：**结束**9：使用 JCuda 将设备内存中的数据转移到主机内存 10：释放所有图形处理器内存

s_{old}, dn, s_{new} $>$出现在路径 P_{min}中。例如，考虑以下包含两条边的示例 P_{min} $<$ s_1 x s_2 $>$ $<$：„;„。在这里，s_2 y s_3 $>$ x s_1 需求单元会从 转移到 s_2，而需求单元 y会从 s_2转移到 s_3。显然，这两项操作彼此完全独立，可以并行执行。不过，本研究没有探讨这一点，而是打算在线程中应用稍复杂的步骤，即更新 ΓDS（接下来将对此进行说明）。

更新分配后，我们还需要更新涉及 P_{min} 的服务中心的 ΓDS。再次考虑前面提到的包含两条边的示例 P_{min}： $<s_1, x, s_2>$；$<s_2, y, s_3>$。对于第一条边，即从 s_1 移动到 s_2 的 x，需要执行以下操作。需要从 ΓDS 中与 s_1 对应的 $k-1$ 个堆中移除需求节点 x，然后将其添加到与 s_2 对应的 $k-1$ 个堆中。请注意， k 是给定问题中的服务中心数量。对这些 $2k-2$ 个堆的所有更新操作彼此独立，并通过线程并行执行。对于 P_{min} 中的所有其他边，也重复类似的过程。

4 分析性评估

4.1 理论分析

在本小节中，我们现在证明能够确立算法2正确性的关键引理。正确性取决于步骤7和步骤14之后不变量的正确性。

对于步骤 7：我们在 Γ(A2) 中检测最负环；如果该环的成本小于零，就修改分配以消除该环。得到的分配为 A3，我们证明 Γ(A3) 不包含任何负成本环。

对于步骤 14：我们在 Γ(A3) 中检测从 s_j 出发的最负路径；如果该路径的成本小于零，就修改分配以消除该路径。得到的分配为 A4，我们证明 Γ(A)4 不包含任何负成本环。

引理1。 考虑算法2步骤6开始时的分配$\mathcal{A}1$ 。如果 $\Gamma(\mathcal{A}1)$ 不存在任何负成本环，那么$\Gamma(\mathcal{A}3)$ （在步骤7结束时）也不存在任何负成本环。

证明策略：添加到 Γ(A1)（以创建Γ(A2)）的唯一新边是由于 d_i的添加而产生的，并且这些边均从s_j出发。这些边将 d_i移动到其他每个服务中心。因此，在 Γ(A2)中引入的任何负环都必须包含一条从s_j出发的边，否则这将意味着在第5步开始时 Γ(A1)中已有负环。令 P_{min} 为由负环细化算法（算法1）确定的、从 w_{out} （s_j的出副本）到 w_{in} （s_j的入副本）之间的路径。

令 C_{min} 为在 Γ(A3)中考虑 P_{min} 里的边，并将s_j 的“入”和“出”副本（w_{in} 和w_{out}，见算法1）合并后形成的环。注意，该环必然存在，这是显而易见的，因为 $NCyc$（Γ(A) s_j）本质上是通过将s_j拆分为“入”和“出”副本而构造的。还要注意，如果 C_{min}的成本大于零，那么该命题显然成立。因此，我们假设 C_{min}的成本小于零。

总的来说，我们采用反证法证明引理：也就是说，如果 Γ(A3)中存在一个负环（比如 C_{neg}），那么 Γ(A1)中必定曾存在一个负环。在详细介绍证明之前，我们先介绍一些基本概念（以及一个示例）。

我们的证明利用来自 C_{min} 和 C_{neg} 的边，在 Γ(A1)中找到一个环 C_{cont}，其总成本小于零。 C_{min} 由 Γ(A2)中的边构成，而 C_{neg} 由 Γ(A3)中的边构成。我们分两步找到 C_{cont}：

（1）消除边依赖关系：如果两条边 e_1 和 e_2的形式分别为 $e_1 = (s_p, s_q, d_x)$ 和 $e_2 = (s_q, s_r, d_x)$，也就是说， e_1 将 d_x从 s_p转移到s_q，而 e_2 将同一需求单元 d_x从s_q转移到 s_r，则称这两条边相互依赖。请注意，只有当 e_1 和 e_2 属于不同的分配子空间多重图时，才会产生依赖关系。通过移除 e_1 和 e_2，并添加边 $e_3 = (s_p, s_r, d_x)$，即可消除这种依赖关系。请注意， e_3 的成本等于 e_1 和 e_2的成本之和，也就是说，消除依赖关系后，总成本保持不变。为找到 C_{cont}，我们取 C_{min} 和 C_{neg} 中的边，并消除所有依赖关系。消除依赖关系后，所得边集具有这样的性质：任一需求单元 $d_x\,(x \neq i)$至多移动一次，且其移动起点是 A1 中分配给 d_x 的服务中心节点。因此，除 C_{min} 中将 d_i 移出s_j的新边外，所有这些边都属于 Γ(A1)。

(2) 找出并排除一条包含新边的环，该环将 d_i移出 s_j。 如果排除这条环上的边，那么所有剩余的边都必须来自 Γ(A1)。正如我们将看到的，负环 C_{cont} 可以在这些剩余边中找到。

图3展示了我们证明策略的一个示例。 S^* 是新增需求单元 d_i所加入的服务中心。在图中， C_{neg} 中的边$(S1, Sa)$和$(S4, Sd)$分别依赖于属于 C_{min}的边$(S^*, S1)$和$(S3, S4)$。在消除这些依赖时，我们加入边(S^*, Sa)和$(S3, Sd)$。这样便得到两个独立环：一个内环 $S1 \to S2 \to S3 \to Sd \to S1$，以及一个外环，$S^* \to Sa \to Sb \to Sc \to S4 \to$

$S5 \rightarrow S^*$。移除将 d_i 移至 $S1$ 的边$(S^*, S1)$，并引入边(S^*, Sa)，将 d_i 移至 Sa。内环 $S1 \rightarrow S2 \rightarrow S3 \rightarrow Sd \rightarrow S1$ 中的所有边都属于 Γ(A1)。外环 $S^* \rightarrow Sa \rightarrow Sb \rightarrow Sc \rightarrow S4 \rightarrow S5 \rightarrow S^*$ 包含唯一一条移动 d_i的边，且这些边都存在于Γ(A2)中。

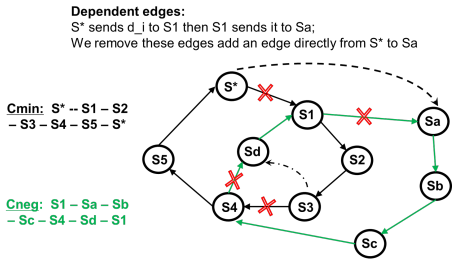


图3：示例说明了用于证明引理1的 C_{min} 和 C_{neg} 环（最佳方案以颜色标示）。

因此，由于 C_{min} 是 Γ(A2) 中成本最低的环，外环的成本必须大于或等于 C_{min}的成本。请注意，依赖消除过程保持总成本不变，因此内环 $S1 \rightarrow S2 \rightarrow S3 \rightarrow Sd \rightarrow S1$的总成本必须小于或等于 C_{neg}的成本，即必须为负。因此，内环就是我们要找的 C_{cont}。

证明。引理1的证明细节。注意，C_{min} 恰好包含一条从 s_j 引出的新边。C_{min}中的任意两条边都相互独立，C_{neg}中的任意两条边也相互独立。考虑由 $C_{min} \bigcup C_{neg}$ 定义的有向图 $H(V, E)$。由于 H是两个环的并集，它满足以下性质：$\forall v \in V, indegree(v) = outdegree(v)$。$H$中会有一些边对（$e_1 \in C_{min}, e_2 \in C_{neg}$）彼此依赖。令 $H'(V, E')$ 表示在 H 中消除依赖后得到的有向图。对于任意依赖，对应于服务中心 s_q 的顶点 v_q 必须同时属于两个环，即 $indegree(v_q) = outdegree(v_q) = 2$。消除依赖后，我们有 $indegree(v_q) = outdegree(v_q) = 1$。因此，$H'$也满足性质 $\forall v \in V$, $indegree(v) = outdegree(v)$，也就是说，$H'$可以分解为有向环。注意以下断言：断言1：$H'$中的所有边都属于 Γ(A2)。只有当 C_{neg} 包含一条依赖于 C_{min}中的某条边的边时，这一性质才可能被违反。由于所有依赖都已消除，该断言成立。

断言2： 中的所有边 H'都属于 Γ(A1)，但从 s_j 引出的新边除外。C_{min}中的边均来自Γ(A1)，但新边除外。C_{neg}中不属于Γ(A1)的任何边都依赖于 C_{min}中的某条边；当我们消除依赖时，所有这些边都会被Γ(A1)中的边替代。因此，只剩下一条从s_j 引出且不属于 Γ(A1)的新边。

断言3： 中所有边的总成本 H'小于 C_{min}中的边的总成本。C_{neg}中的边的总成本小于零，并且依赖消除保持成本不变。

为完成论证，找出包含从 s_j 出发的新边的成本最低的环 $C1$。该环的成本必须大于或等于 C_{min} 的成本，因为 C_{min} 是 Γ 中成本最低的环（A2）。因此，$H' - C1$ 的成本小于零，也就是说，将其分解为有向环后，至少得到一个负成本的环。由于 $H' - C1$ 中的所有边都属于 Γ(A1)，这个负环已存在于 Γ(A1) 中。□

引理2。 $\Gamma(\mathcal{A}4)$ 在算法*2*第*14*步结束时不含任何负成本环

证明。设 P_{min} 是在 $NPath\Gamma(A3, s*)$中 $s*$的“入”和“出”副本（$s*_{in}$和 $s*_{out}$）之间的最小成本路径。P_{min}由一直延伸到某个服务中心节点（设为）s_l的需求单位转移边，以及随后从 s_l 到 $s*_{in}$的惩罚转移边组成。设 $C*$ 为在 P_{min}中将 $s*_{in}$ 和 $s*_{out}$ 合并后形成的环。$C*$ 中的所有需求单位转移边都属于 Γ(A3)。反设存在负环 C_{neg}，且 Γ属于 H(A4)。将有向图 $H(V, E)$定义为 $C* \bigcup C_{neg}$，消除 H中的依赖后所得的有向图记为 H'。H'中的所有边，除惩罚转移边外，都属于 Γ(A3)。设 $C1$ 为 H'中包含从 s_l 到 $s*$的惩罚转移边的最小成本环。$C1$ = 为从 $s*$ 到 s_l + 的路径，其中包含从 s_l 到 $s*$的惩罚转移边。与引理1的证明类似，我们断言，有向图$H' - C1$ 仅包含来自 Γ(A3)的边，总成本小于零，并且可以进行有向环分解。因此，它必然包含一个负环，该负环也存在于 Γ(A3)中。□ □

4.2 空间与时间复杂度分析

定理1 一个*LBDD*实例，算法*2*的运行时间为$O(nk^3 + nk^2 \log n)$ *time*，使用 $O(nk)$*space*，其中 n是需求节点的数量，k是服务中心的数量。

详细证明见附录C。

5 实验分析

数据集：我们的实验使用了印度新德里的道路网络。该道路网络包含65,000个节点，数据来自OpenStreetMaps [www.openstreetmap.org]。在实验中，我们从图中随机选取了一些顶点，并将其设为服务中心。其余顶点则指定为需求顶点（每个顶点的单位需求为1）。道路网络上的最短距离表示将一个需求单元分配给一个服务中心的成本。我们预先计算了所有需求单元与服务中心之间的最短路径距离。

实验中根据比率 n：k（称为服务中心比例）调整服务中心的数量。其中，n表示需求单元的数量，k表示服务中心的数量。比率 600 : 1 表示每 600 个需求顶点对应一个服务中心。对于给定数量的服务中心和需求，所有服务中心的总容量为 $\theta * n$。该总容量会随机分配给 k服务中心。为便于解释结果，我们不生成单调函数，而是假设每个服务中心都有一个在某个范围内随机选取的惩罚值。实验中对该范围进行了调整。

算法使用 Java 1.8 实现。我们在一台 Ubuntu 机器上进行了实验，该机器配备 NVIDIA Tesla P100 图形处理器（3584 个 CUDA 核心）和 Intel Xeon Silver 4110 中央处理器（2.10GHz）。我们的 ASRAL 算法在主内存中使用了约 16GB RAM。para-ASRAL 在图形处理器上额外使用了 8GB

比率	# 需求单元	# 服务中心
500:1	65771	131
600:1	65793	109
700:1	65808	94
800:1	65820	82
900:1	65829	73

表2：不同服务中心比例值对应的需求单元数和服务中心数。

内存。

候选算法： 实验中比较了以下 4 种算法。

(1) **我们提出的 ASRAL 算法和 Para-ASRAL 算法**。(2) **LoRaL 算法 [12]** 我们将 LoRaL 的"探索参数"（ k 中的值 [12]）设置为服务中心的数量（即 $|S|$ ）后运行。这意味着 LoRaL 会探索 $|S|$ 条用于重组的潜在路径（针对当前部分解），然后选择最佳路径。请注意，这是 LoRaL 算法在任何 LBDD问题实例上能够进行的最大探索量。换句话说，**本节报告的 LoRaL 解质量是 LoRaL 能达到的最佳水平**。(3) **最小成本二分图匹配：** [3]中给出的过程。

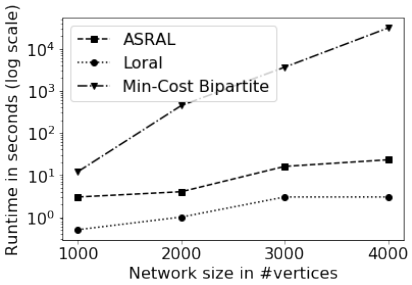
可变参数： 以下是我们实验中所改变参数的详细信息。

(1)在我们的实验中，总容量参数值 θ 为 0.3（低总容量）或0.7（高容量）。（2）在我们的实验中，所有服务中心的惩罚值要么从范围 [1 到 200]（低惩罚值）中选取，要么从范围 [200 400]（高惩罚值）中选取。（3）服务中心比例（n： k）：我们选择了以下五个比率值：500:1、600:1、700:1、800:1 和 900:1。表2描述了对于不同服务中心比例值，我们数据集中的需求单元和服务中心数量。

我们将结果分为两部分。首先，我们比较 ASRAL 算法与其他方法（最小成本二分图匹配和 LoRaL）的运行时间和解质量。随后，我们展示 para-ASRAL 算法的实验分析，并将其与 ASRAL 和 LoRaL 算法进行比较。请注意，ASRAL 和 para-ASRAL 的解质量相同。

5.1 ASRAL 与相关工作的比较

ASRAL 与最小成本二分图匹配的比较： 图 4 展示了这些结果。我们的结果（图 4(a)）表明，**最小成本二分图匹配的最优算法无法扩展到包含 4000 个顶点以上的图**。其主存需求增长也非常快。对于包含 4000 个需求顶点和 40 个服务中心的 LBDD 问题实例，该算法需要超过 90GB 的 RAM。因此，在大型问题实例（规模为 65000 个节点的道路网络）上的实验中无法纳入该最优算法。



（a）执行时间

#Demand Units	LoRaL Algorithm	ASRAL Algorithm	Min-Cost Bipartite Matching
1000	106814	106759	106643
2000	177893	177623	177161
3000	355455	354456	349861
4000	358604	357205	355524

（b）目标函数值

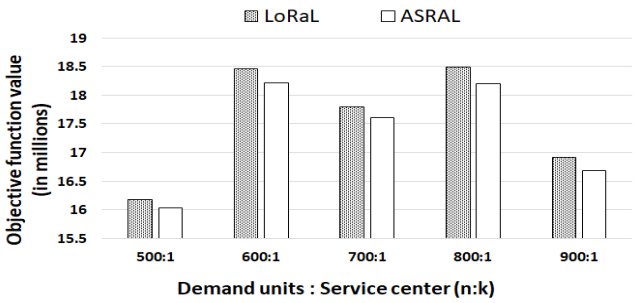
图 4：ASRAL 与最小成本二分图匹配算法的比较。n： k **为** $300:1$

ASRAL 与 LoRaL 的比较： 图 5 和图 6 展示了这些结果。**ASRAL 得到的目标函数值显著更低**。如图 5 所示，我们提出的 ASRAL 得到了显著更优的解，在某些情况下（见图 5(d)），其目标函数值低了 *50* 万（甚至更多）个单位。总体而言，在以下两种情况下差异更大：(a) 服务中心的惩罚值较高（介于 200 至 400 之间）；和/或 (b) 服务中心数量较少（参见图 5 中 |D|:|S|=800:1 和 900:1 的数据）。ASRAL 的执行时间确实长于 LoRaL（图 6），但需要注意的是，执行时间的增加带来了显著更低的目标函数值。ASRAL 能够扩展到大型问题实例，而最小成本二分图匹配的最优算法则无法做到这一点。

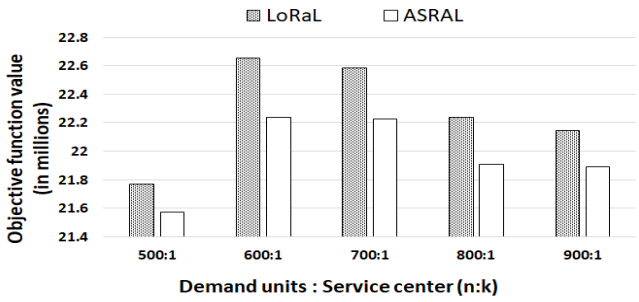
分析 ASRAL 的运行时间： 在图 7 中，我们展示了 ASRAL 算法执行时间的典型分解情况。该图表明，74.15% 的执行时间用于更新 ΓDS。Bellman-Ford 算法占用了 6.44% 的执行时间。其他所有操作总共占用了 19.41% 的时间。上述观察促使我们使用线程更新 ΓDS，并使用图形处理器核心执行 Bellman-Ford 算法，从而实现 ASRAL 算法的并行版本。这些思路分别在第 3.6.1 节和第 3.6.2 节中进行了讨论。

5.2 比较 ASRAL 算法与 Para-ASRAL 算法

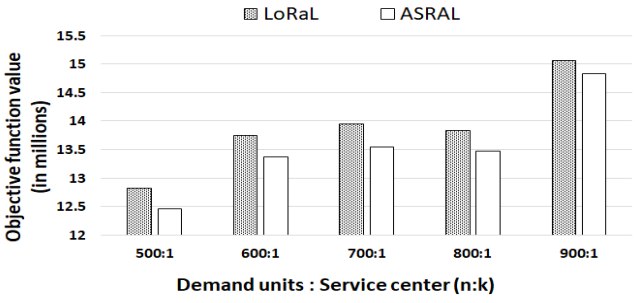
并行化 Bellman-Ford 算法的效果： 图8比较了 ASRAL 和 Para-ASRAL 算法中 Bellman-Ford 算法（用于负环和负路径细化过程）所耗费的总时间。如前所述，Para-ASRAL 使用图形处理器代码并行化 Bellman-Ford 算法。结果表明，借助图形处理器核心，我们能够大幅缩短 Bellman-Ford 算法的运行时间。我们还观察到，在服务中心



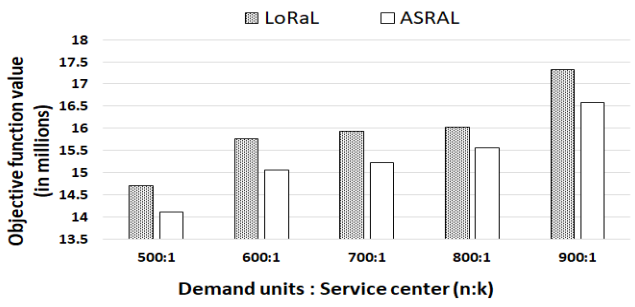
(a) 容量 = 0.3，惩罚范围为 1 到 200



(b) 容量 = 0.3 和惩罚范围从 200 到 400

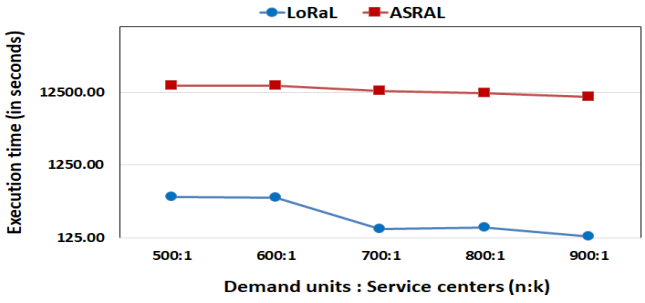


(c) 容量 = 0.7 和惩罚范围从 1 到 200

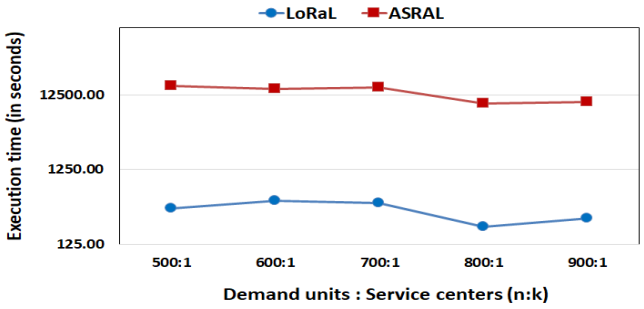


(d) 容量 = 0.7，惩罚范围为 200 至 400

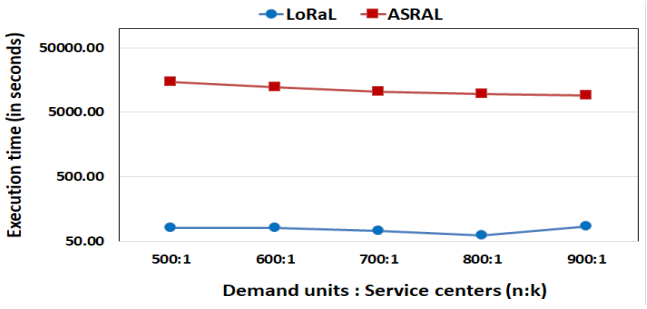
图 5：LoRaL 和 AS- RAL 算法的最终目标函数值



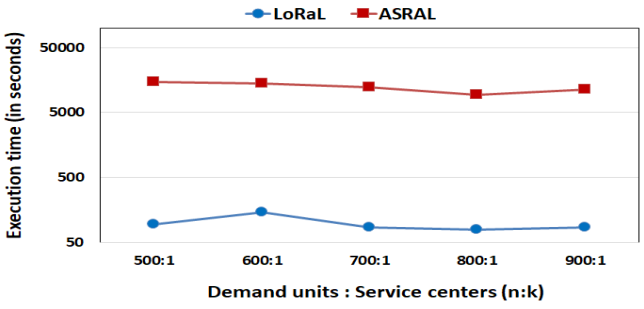
(a) 容量 = 0.3，惩罚范围为 1 至 200



(b) 容量 = 0.3，惩罚范围为 200 至 400



(c) 容量 = 0.7，惩罚范围为 1 至 200



(d) 容量 = 0.7，惩罚范围为 200 至 400

图 6：LoRaL 和 ASRAL 算法执行时间对比（以 log10 为刻度）

比例较低时，运行时间的缩短幅度更大（例如图8中 500:1 和 600:1 对应的数值）。这是因为在我们的数据集中，随着服务中心比例的提高，分配子空间多重图的规模会减小。

例如，当服务中心比例为 900:1 时，我们的数据集中只有 73 个服务中心。而在比例为 500:1 时，我们有 131 个服务中心。

并行化的效果 Γ*DS* 更新： 图 9 展示了 ASRAL 和 para-ASRAL 算法更新 Γ*DS* 所花费的总时间。如前所述，para-ASRAL 使用 CPU 核心并行更新 Γ*DS*。结果表明，在不同的总容量、惩罚和服务中心比率下，我们至少能够将 Γ*DS* 的更新时间缩短 85%。

Para-ASRAL 算法实现的总加速比： 在图 10 中，我们比较了 ASRAL、para-ASRAL 和 LoRaL 算法的总运行时间。实验结果表明，在不同的总容量、惩罚和服务中心比率下，Para-ASRAL 的运行时间至少比 ASRAL 算法少 78%。Para-ASRAL 的运行时间与 LoRaL 算法更为接近。

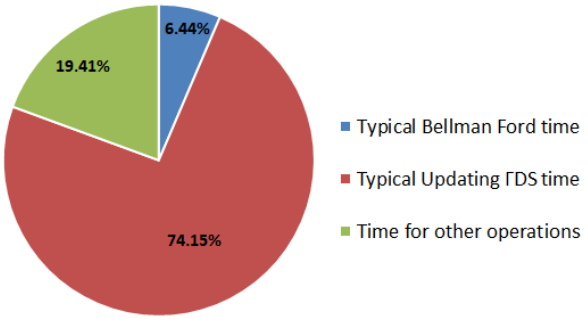
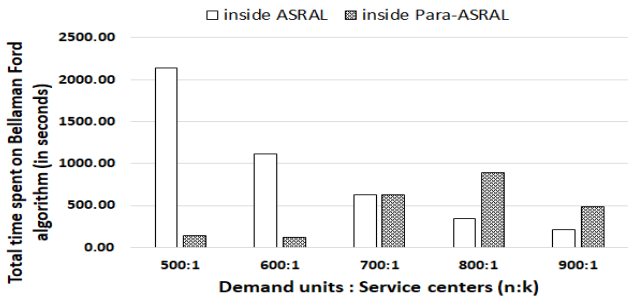
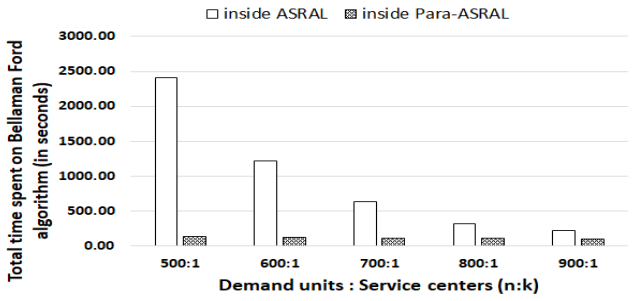


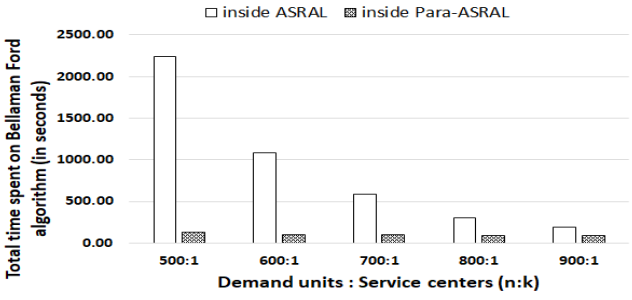
图 7：ASRAL 算法执行时间分解



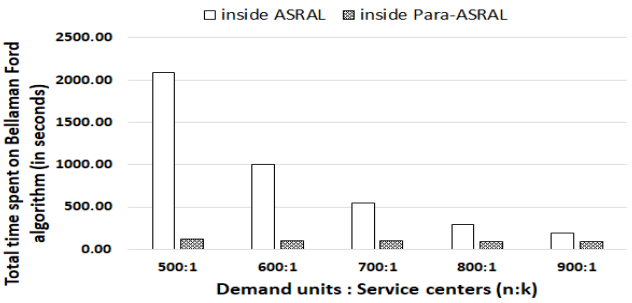
(a) 容量 = 0.3 和惩罚范围 (1 200)



(b) 容量 = 0.3 和惩罚范围 (200 400)

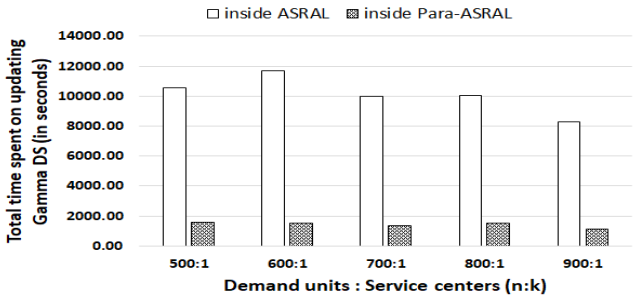


(c) 容量 = 0.7 和惩罚范围 (1 200)

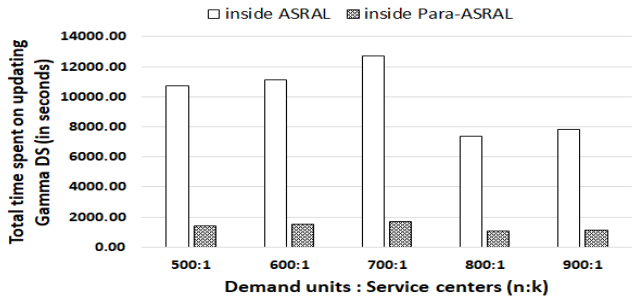


(d) 容量 = 0.7 和惩罚范围 (200 400)

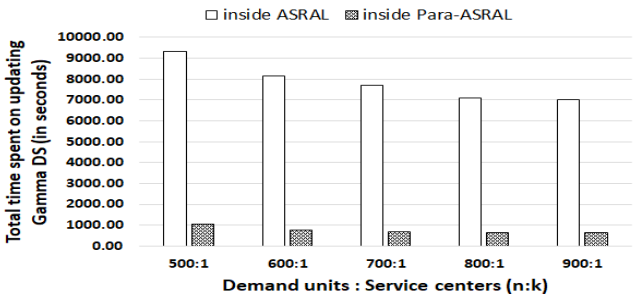
图 8：ASRAL 和 para-ASRAL 算法中 Bellman-Ford 算法的总耗时。



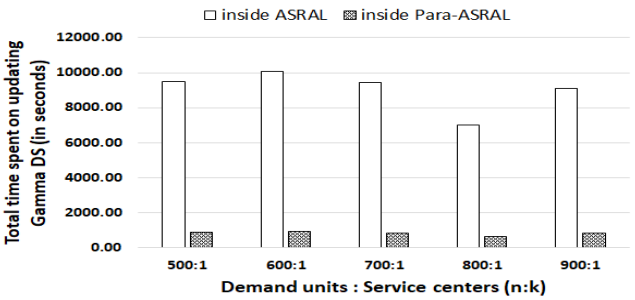
(a) 容量 = 0.3，惩罚范围 (1 200)



(b) 容量 = 0.3，惩罚范围 (200 400)

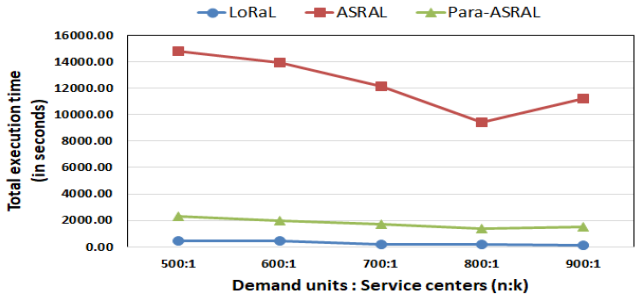


(c) 容量 = 0.7，惩罚范围 (1 200)

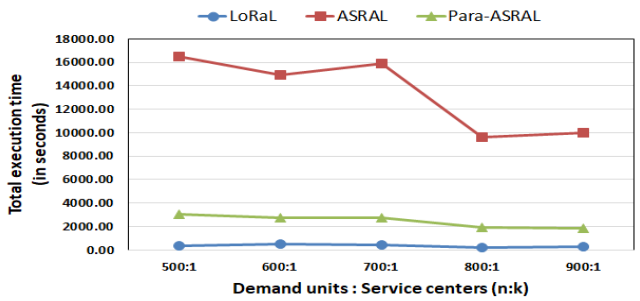


(d) 容量 = 0.7，惩罚范围 (200 400)

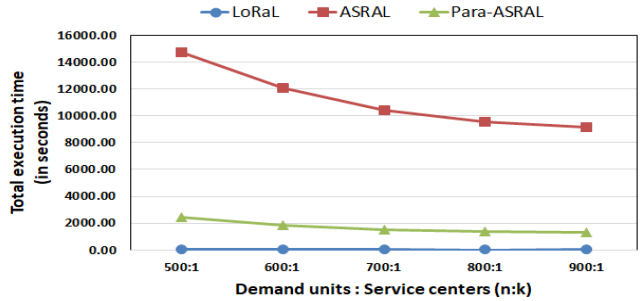
图 9：ASRAL 和 para-ASRAL 算法中更新 ΓDS 所花费的总时间



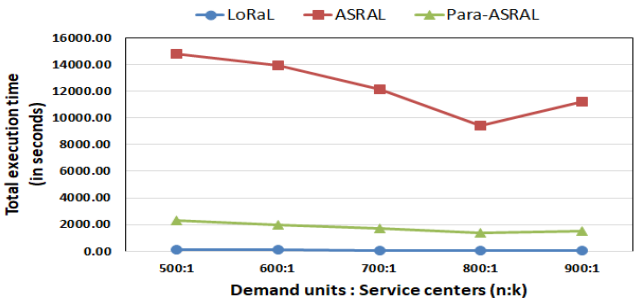
(a) 容量 = 0.3，惩罚范围 (1 200)



(b) 容量 = 0.3，惩罚范围 (200 400)



(c) 容量 = 0.7，惩罚范围 (1 200)



(d) 容量 = 0.7，惩罚范围 (200 400)

图 10：LoRaL、ASRAL 和 para-ASRAL 算法的总执行时间

6 结论

负载均衡需求分配（LBDD）是一个具有重要社会意义的问题。LBDD问题可以归约为最小费用二分图匹配问题。然而，最小费用二分图匹配问题的最优算法无法扩展到大型图。我们提出的ASRAL算法能够实现扩展，同时相较于其他方法保持更高的解质量。我们还为所提出的ASRAL 算法开发了并行实现（para-ASRAL），以提高其可扩展性。para-ASRAL的速度显著快于ASRAL，同时保持相同的解质量。

致谢： 由于投稿采用双-盲审稿，稍后补充。

作者贡献： 由于投稿采用双盲评审，稍后补充。

参考文献

[1] F. Aurenhammer。1987。幂图：性质、算法与应用。*SIAM*计算杂志 16，1（1987），78–96。[2] H. Bast 等。2007。带有换乘节点的道路网络快速路由。《科学》316，5824（2007），566–566。[3] Edward Bortnikov、Samir Khuller、Jian Li、Yishay Mansour 和 Joseph Seffi Naor。2012。负载-距离均衡问题。《网络》59，1（2012），22–29。[4] Robert G Busacker 和 Thomas L Saaty。1965。有限图与网络。麦格劳-希尔。[5] Thomas H. Cormen、Clifford Stein、Ronald L. Rivest 和 Charles E. Leiserson。2001。《算法导论》（第2版）。麦格劳-希尔高等教育出版社。[6] Daniel Delling 等。2013。PHAST：硬件加速的最短路径树。《并行与分布式计算杂志》73，7（2013），940–952。[7] U. Demiryurek 和 C. Shahabi。2012。网络沃罗诺伊图索引。在第*17*届高级应用数据库系统国际会议上。施普林格，526–543。[8] Ali Diabat。2016。具有单一供货的容量约束设施选址与库存管理问题。《优化快报》10，7（2016），1577–1592。[9] Robert Geisberger 等。2008。收缩层次：道路网络中更快、更简单的分层路由。在*WEA'08*会议论文集中。施普林格-弗拉格，319–333。[10] Andrew V. Goldberg、Haim Kaplan 和 Renato F. Werneck。2007。触手可及的更优地标。在*WEA'07*会议论文集中。施普林格-弗拉格，38–51。[11] Jing 等。1998。用于路径查询处理的分层编码路径视图：最优模型及其性能评估。《*IEEE*知识与数据工程汇刊》10，3（1998），409–432。[12] Ankita Mehta、Kapish Malik、Venkata M. V. Gunturi、Anurag Goel、Pooja Sethia 和 Aditi Aggarwal。2018。过载惩罚下网络沃罗诺伊图中的负载均衡。在数据库与专家系统应用中。457–475。[13] A. Okabe 等。2008。广义网络沃罗诺伊图：概念、计算方法与应用。《国际地理信息系统杂志》22，9（2008），965–994。[14] Leong Hou U 等。2008。空间数据库中的容量约束分配。在数据管理国际会议（*SIGMOD*）论文集中。15–28。[15] Leong Hou U 等。2010。容量约束下空间数据集的最优匹配。《*ACM*数据库系统汇刊》35，2（2010），9:1–9:44。[16] Xiaokui Xiao、Bin Yao 和 Feifei Li。2011。道路网络数据库中的最优位置查询。在第*27*届数据工程国际会议（*ICDE*）论文集中。804–815。[17] Yonghong Yan、Max Grossman 和 Vivek Sarkar。2009。JCuda：使用 CUDA 加速 Java 程序的程序员友好型接口。887–899。https: //doi.org/ 10 1007 978 3 642 03869 3_82。 / - - - - KwangSoo Yang 等。容量[18] 2013约束网络沃罗诺伊图：研究结果总结。在*SSTD 2013*会议论文集中。56–73。[19] K. Yang 等。2015。容量约束网络沃罗诺伊图。《*IEEE*知识与数据工程汇刊》27，11（2015年11月），2919–2932。[20] Bin Yao 等。2014。道路网络数据库中最优位置的动态监测。《*VLDB*期刊》23，5（2014），697–720。

附录A 负路径优化

算法4 负路径细化**输入：** （a）当前分配 A；（b）锚定服务中心 SC_θ **输出：** 调整后的分配 F。移除输入 A 中从 Sc_θ 开始、位于 Γ(A)中的负路径。1：创建 NegPath 分配子空间图 $NPath\Gamma$(A SC_θ)，并将 SC_θ 设为锚点顶点。2：令 w_{out} 和 w_{in} 分别为锚点顶点 SC_θ 在 $minPath\Gamma$(A)中的"IN"和"OUT"副本。3：从 w_{out} 到 w_{in} 查找 P_{min} 中成本最低的路径 $NPath\Gamma$(A)。4：**如果** $P_{min} < 0$ 的成本**则**5：**对于** 每个需求节点转移< s_p、 dn、 $s_q >\in P_{min}$ **执行**6：F ← 通过从 s_p 中移除 dn 并将其添加到 s_q 来更改 A。7：更新 ΓDS8：A ←F 9： P_{min} 中对应边的成本 10：**结束循环**11： P_{min} 中最后一条边的成本/*此边表示惩罚值之差。12：**结束条件**13：返回新分配 B

附录 B 严格容量约束下 LBDD 的最优解

我们之前提出的用于求解 LBDD 的算法可以轻松调整，以便在不允许服务中心超载时给出最优解。我们将这一变体称为 *Strict-LBDD*问题。以下是该问题的形式化定义。

严格LBDD问题

输入:

(1) 一组 S 个 k 服务中心。(2) 一组 D 个 n 需求单元。(3) 一个需求-服务成本矩阵 CM。CM(i, j) 包含将需求单元 d_i 分配到服务中心 s_j $(\forall d_i \in D, \forall s_j \in S)$ 的成本（正实数值）。(4) 每个服务中心 $s_i \in S$的容量（c_{s_i}）（整数）。**输出**：使以下目标函数最小化的完整分配。

$$\textbf{Minimize}\left\{\sum_{\substack{s_i \in Service\\ Centers}}\left\{\sum_{\substack{d_j \in Demand\ unit\\ allotted\ to\ s_i}} CM(d_j, s_i)\right\}\right\} \quad (2)$$

约束条件： (i) 一个需求单元只能分配给一个服务中心；(ii) 任何服务中心分配到的需求不得超过其容量。

可以想象，如果存在**超额需求**，即$n > \Sigma_{i=1}^{i=k}(c_{s_i})$，那么就无法生成完整分配。在这种情况下，我们按如下方式修改问题实例：(1) 新增一个容量为 $n - \Sigma_{i=1}^{i=k}(c_{s_i})$的服务中心 s_{k+1}。

(2) 令 max 为给定 CM 中的最大值。我们通过添加一列来修改 CM，该列对应服务中心 s_{k+1}，并将这一列中的所有值设为 $max+1$。

请注意，此修改会使目标函数的值增加 $(n-\Sigma_{i=1}^{i=k}(c_{s_i}))\times(max+1)$。由于这一增量适用于所有可能的完整分配，因此不会影响最优解。

B.1 从 Γ() 的角度刻画最优解

为了刻画*strict-LBDD*的最优解，我们假设在分配过程的任意阶段，所有服务中心始终通过使用某些虚拟需求单位而达到满负荷。这些虚拟需求单位在所有服务中心的分配成本均为零。因此，如果 s_i 有 p 单位的剩余容量（用于容纳输入中给出的"实际"需求单位），那么由于每个 p 虚拟需求单位的转移，Γ(A)将有 p 条从 s_i 到任意s_j 的零成本边。

如前所述，Γ(A) 中的负成本环有助于提高解质量（即降低目标函数值）。下面的定理（定理 2）形式化了 *Strict-LBDD* 解的最优性与负环存在性之间的关系。文献中也可以找到类似结果（例如参见 [4] 中的定理7.8），但为完整起见，我们在此给出详细说明。在证明中，我们将使用以下命题，该命题是图论中的常识。

命题1。一个有向图 $G(V,E)$ 可以分解为有向环，当且仅当 $\forall v\in V, indegree(v)=outdegree\ v$。

定理 2。设 $\mathcal{A}$ 是 Strict-LBDD 问题的一个分配，Γ（A）是相应的分配子空间多重图。$\mathcal{A}$ 是最优解，当且仅当 Γ（A）不包含任何简单负成本环。

证明。我们在所有论证中都假设环是简单的。很容易证明，如果某个分配包含一个非简单负环，那么它也包含一个简单负环。

由于移除一个负成本环会降低总成本，因此显然，不存在负成本环是最优性的必要条件。现在我们将证明，不存在负成本环也是最优性的充分条件。

设 A^{opt} 是一个最优分配，A 是任意一个分配，且 Γ(A) 不包含任何负成本环。我们将证明，如果 $Cost(A^{opt})$ 小于 $Cost(A)$，那么 Γ(A) 必定包含一个负成本环。

我们按如下方式构造 A 和 A^{opt} 的差异多重图，记作 $DM(A^{opt},A)$：(1) 对每个服务中心 s_i，$1\le i\le k$（或在超额需求的情况下为 $k+1$），向 $DM(A,A^{opt})$ 中添加一个顶点 v_i。(2) 设 d_i，$1\le i\le n$，是任意一个需求单元，使得 A 将 d_i 分配给服务中心 s_p，而 A^{opt} 将 d_i 分配给服务中心 s_q，且 $p\neq q$。$DM(A,A^{opt})$ 包含一条边 (v_p,v_q,d_i)，其权重为 $CM(d_i,s_q)-CM(d_i,s_p)$。

请注意，$DM(A,A^{opt})$ 表示将分配A 转换为 A^{opt}所需的所有需求单元移动。因此，所有这些边都必须属于 Γ(A)。

命题。我们可以通过适当地添加转移虚拟需求单位的边，确保 v_i 在$DM(\mathcal{A}、\mathcal{A}^{opt})$ 中的出度与 v_i的入度相同。

引理证明。在构造 $DM(A,A^{opt})$ 时，如果从 s_i 移出的需求单元数量与移入 s_i 的需求单元数量相同，则无需证明。令 δ_i 表示移入 s_i 的需求单元数量与移出 s_i 的需求单元数量之差。如果 $\delta_i>0$，我们将在 v_i 添加 δ_i 条成本为零的出边，以便从 s_i 移出 δ_i 个虚拟需求单位。注意，由于 A 和 A^{opt} 都是有效分配，因此可以保证存在这些虚拟需求单位。相应地，如果 $\delta_i<0$，我们可以在 v_i 添加 δ_i 条成本为零的入边。由于 A 和 A^{opt} 的需求单元数量相同，$\Sigma(\delta_i)=0$。这意味着，虚拟需求单位转移弧的出弧总数等于入弧总数。由于这些弧的成本均为零，因此我们可以任意用权重为零的边连接这些顶点，以确保该结论成立。**引理证毕**

因此，根据命题1，$DM(A,A^{opt})$可以划分为一组边不相交有向环。由于A 的成本高于 A^{opt}，$DM(A,A^{opt})$中所有边的权重之和必定小于零，因此这些环中至少有一个环的成本小于零。所有这些边都必须属于 Γ(A)，即 Γ(A) 具有一个负环。□

B.2 严格LBDD问题的最优算法

算法5提出了一种针对严格LBDD问题的最优算法。结合引理1和定理 2，可以轻易证明算法5的最优性。

附录C 其他详细证明

定理1的证明：为每个需求单元确定"最合适"（使用 CM）的服务中心需要 $O(nk)$ 时间。随后，ASRAL 算法构建最小距离堆，还需要 $O(n)$ 时间。注意，初始时 ΓDS 为空。因此，我们将其在算法执行过程中的更新成本计入（即步骤 4 至 17 之间）。

ASRAL 算法的主循环（步骤4至17）执行 n 次。在此循环中，提取最小值 操作（步骤5）耗时 $O(\log n)$。在步骤6中，我们将 d_i 分配给 s_j，这需要更新 $k-1$ 个堆，因此最多需要 $O(k\log n)$ 次操作。步骤7调用负环细化过程（算法1）。在算法1中，步骤1用 $O(k^2)$ 步构造 $NCyc\Gamma()$。步骤3用 $O(k^3)$ 步找到 P_{min}，步骤5中的循环最多执行 k^2 次堆操作，每次操作耗时 $O(\log n)$ 步。因此，算法1的运行步数为 $O(k^3+k^2\log n)$。负环细化之后，ASRAL 算法在步骤14执行负路径细化（使用算法4）。与负环细化类似，负路径细化（算法4）也需要 $O(k^3+k^2\log n)$ 步。

综上，ASRAL 第4步至第17步中的整个 while 循环可以在 $O(nk^3+nk^2\log n)$ 时间内完成。因此，ASRAL（算法2）的总时间复杂度为 $O(nk^3+nk^2\log n)$。

关于空间复杂度，ΓDS 最多会占用 $O(nk)$ 空间，因为在执行的任何阶段，每个需求单元最多只能存在于 k 个堆中（用于实现 ΓDS）。

算法5 严格 LBDD 的基于子空间重新调整的方法**输入：** (a) 需求-服务成本矩阵 CM；(b) 一组 S 服务中心。对于每个 $s_i \in S$，其容量为 c_{s_i}；(c) 一组 D 需求单元。**输出：** 该问题实例的最优分配 1：如果总容量小于总需求，则添加虚拟服务中心 s_{k+1}，并通过为 s_{k+1} 添加一列来修改 CM，将所有值设为（CM 中的最大分配成本 +1） 2：初始化 A1。创建虚拟需求单位，并以零成本将其分配给所有服务中心 $s_i \in S$，使每个服务中心的容量均达到满载。 3：**当**存在未访问的需求单元**时执行**4：选取一个未访问的需求单元 d_i。将 d_i 标记为已访问。 5：$X \leftarrow$ 具有剩余容量的所有服务中心集合（即仍有一些虚拟需求节点分配给它们）。 6：从 X 中确定一个服务中心 s_j，使 CM(d_i, s_j) 最小。 7：A2 $\leftarrow$ 通过在 s_j 中用 d_i 替换一个虚拟需求单元来更改 A1 8：A3 $\leftarrow$ 负环细化(A2,s_j)（使用算法1） 9：A1 $\leftarrow$ A3 10：**结束循环**11：撤销预处理步骤后返回所得分配（即移除所有虚拟需求单位，或移除虚拟服务中心 s_{k+1}（及分配给它的需求单位））。

除 ΓDS 外， $NCyc\Gamma()$ 和 $NPath\Gamma()$ 将占用 $O(k^2)$ 空间。最小距离堆最多占用 $O(n)$ 空间。因此，ASRAL 的总空间复杂度为 $O(nk)$。（定理1证明结束）

附录D LBDDINSTANCE，其中学校作为服务中心

在这种情况下，目标函数的一个可行计量单位可以是以美元计的单日“运营”总成本。边成本可以是经过该边时支付的票价金额。对于惩罚，可以采用师生比等指标。如果学校保持 1:F 的师生比，那么每超过学校容量分配一个单位，其超载惩罚就是教师年薪/(F x 365)。通过纳入图书-学生比、实验室设备-学生比等其他参数，可以轻松构建更复杂的惩罚函数。在这种情况下，学校容量应设为按教师总数、设备总量、图书总量等计算出的容量中的最小值。需要注意的是，只要这些惩罚函数符合定义4，所提出的算法就不受其具体实现细节的影响。