**基于三维图建模的数据中心**

**通用线缆路由算法研究**

**目** **录**

**[1 引言](#bookmark2)****[1](#bookmark2)**

**[2 算法设计](#bookmark3)****[2](#bookmark3)**

[2.1 机房情况分析 2](#bookmark4)

[2.2 机房建模方法 4](#bookmark5)

[2.3 最短路由/成本优先策略的实现 5](#bookmark6)

[2.4 双路由策略的实现 7](#bookmark7)

[2.5 负载均衡策略的实现 8](#bookmark8)

**[3 算法实现及验证](#bookmark9)****[11](#bookmark9)**

[3.1 实验环境介绍 11](#bookmark10)

[3.2 项目关键实现 11](#bookmark11)

[3.3 实验结果与分析 16](#bookmark12)

**[4 结论](#bookmark13)****[20](#bookmark13)**

**基于三维图建模的数据中心通用线缆路由算法研究**

**摘要：**随着数据中心机房空间结构与布线复杂度的不断提升，传统的路由规划方法在精度、 灵活度与策略多样性方面已难以满足工程需求。本文提出了一种基于三维图建模的通用线缆 路由算法，通过将机房空间结构抽象为带权无向图，结合 A\*算法、不相交路径和动态权重 等策略，实现了多种目标下的智能路由规划。实验结果表明， 该算法能够高效、准确地处理 复杂的机房环境，同时具备良好的运行效率，即使在较低的硬件配置下仍能保持不错的性能。 本研究为数据中心线缆路由规划提供了更加智能与灵活的解决方案，具有一定的工程应用价 值。

**关键词：**数据中心；线缆路由规划；三维图建模；A\*算法；Python

**Abstract:** With the increasing complexity of spatial structures and cabling requirements in data center equipment rooms, traditional routing methods have become inadequate in precision, flexibility, and strategic diversity. This paper proposes a universal cable routing algorithm based on 3D graph modeling. By abstracting the spatial structure of equipment rooms into weighted undirected graphs and integrating strategies such as the A\* algorithm, disjoint paths, and dynamic weights, the algorithm achieves intelligent routing planning under multiple objectives. Experimental results demonstrate that the algorithm efficiently and accurately handles complex equipment room environments while maintaining robust performance even under low hardware configurations. This study provides a more intelligent and flexible solution for data center cable routing planning, offering significant engineering application value.

**Keywords:** Data center; Cable routing planning; 3D graph modeling; A\* algorithm; Python

**1 引言**

数据中心作为支撑数字经济发展的新型基础设施[1]，在 5G、云计算、人工 智能等新一代信息技术的推动下，正经历着前所未有的变革与发展。这种变革体 现在诸多方面，比如：

（1）空间布局：机房物理结构呈现多样化、复杂化趋势，比如开始出现多 层架构的微模块机房[2]，实现了快速部署与灵活扩展；

（2）功率密度：单机架功率从 5kW 逐步提升至 7kW 、10kW 甚至 20kW，

设备装机密度随之不断提升；

（3）布线复杂度：随着设备装机密度的增加以及布线策略需求的多元化（如 最短路由、成本优先、双路由、负载均衡等），线缆路由规划的复杂性显著提升。

目前行业内对线缆路由的规划方式也正逐步从传统人工模式向工具自动化 方向发展，常见方法是将机房平面布置投影到二维平面，通过坐标计算来确定走 线方向和线缆长度。然而，这样的方法存在明显局限性：

（1）仅能提供粗略的布线方向（如沿列头或列尾布线）；

（2）线缆长度精度不足；

（3）无法适配较为复杂的空间结构，往往需要人工干预调整；

（4）简单的建模方式难以满足日益增长的多元化布线策略需求。

针对上述问题，本文提出一种面向数据中心的通用线缆路由算法。该算法通 过三维图建模抽象机房空间结构，结合 A\*算法、不相交路径和动态权重等策略， 实现了多种目标下的智能路由规划。实验结果表明， 该算法能够高效、准确地处 理复杂的机房环境，可显著提高布线规划的精度与效率，为数据中心线缆路由规 划提供了更为智能和灵活的解决方案。

**2 算法设计**

**2.1 机房情况分析**

目前数据中心机房主要可分为单层和多层（如微模块机房）两种结构。机房 内通常采用 19 英寸标准机架，机架成行排列，根据建筑面积的不同，单个机房 可容纳数十至数百个机架。

线缆通常通过线槽、桥架等装置（以下简称桥架[3] ）进行布置。桥架可分为 列桥架和主桥架两类：

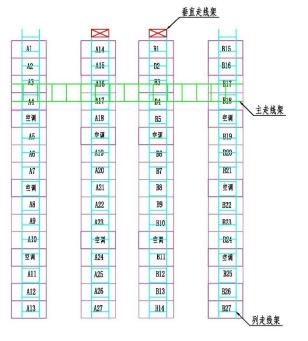
（1）列桥架：沿机柜排布方向布置，主要功能是便于机柜内设备线缆的布 放与管理；

（2）主桥架：沿机房纵向轴线布置，通常与横向延伸的列桥架垂直交叉， 形成网格化拓扑结构，从而实现各列机柜间的互联互通。

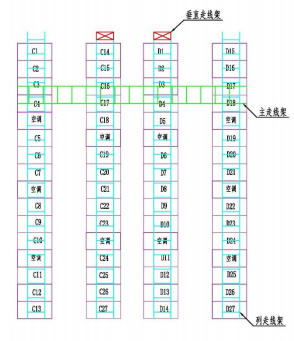
对于多层机房，还会引入垂直桥架以实现不同层间的互通。主桥架和垂直桥 架的数量与位置排布因机房规模、布局和需求而异， 呈现出较高的复杂性和多样 性。这种复杂性不仅体现在桥架的物理布置上， 还体现在其对布线规划、空间利

用和运维管理的影响中。

以下为某运营商近年来开始使用的微模块机房的平面布置示意图，机房整体 为钢框架双层结构，整体在工厂加工生产，然后在现场组装，具有简化建筑形式、 提升装机效率、缩短建设周期等特点。



**图** **1** **微模块一层平面布置图**



**图** **2** **微模块二层平面布置图**

**2.2 机房建模方法**

机房的空间结构相对比较复杂，走线桥架灵活多变的布置方式是其主要原因 之一，随着多层结构机房的出现，布线问题从传统的二维扩展至三维，进一步增 加了规划与管理的难度。

为精确描述机房的空间拓扑关系，本文将机房的空间结构抽象为带权无向图。 通过节点（*node*）和边（*edge*）的方式存储和记录必要的空间信息，并结合权重 控制来实现路由规划的灵活性与精确性。

（1）节点的设计

本文场景下，节点代表线缆路由的起点与终点，通常是机架或机架中的设备， 则每个节点需包含以下必要属性：

**表** **1** **节点的关键属性**

|  |  |
| --- | --- |
| 属性 | 说明 |
| id | 唯一标识符 |
| name | 设备名称，便于识别与检索 |
| coordinates | 三维坐标（*x, y, z*），记录节点在空间中的物理位置信息 |

（2）边的设计

边用于描述节点之间的空间连接关系，需要将复杂的空间关系抽象为边的各 种属性，使得桥架走向可以通过边来表征，每条边需包含以下关键属性：

**表** **2** **边关的关键属性**

|  |  |
| --- | --- |
| 属性 | 说明 |

id 唯一标识符

node\_a/node\_b 记录边所连接的两个节点的信息

path\_type 记录边的类型，不同类型的路径（如：走线架、光纤槽道等） 将对应

不同的物理属性和功能特性

base\_length 该段路由实际的物理距离

redundancy\_factor 冗余系数，用来控制线缆的冗余长度

max\_capacity 最大线缆承载量，设置路径的承载上限

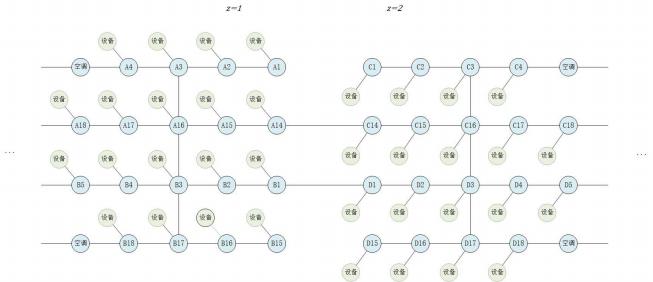
current\_load 当前负载，动态反映路径的使用情况

flags 状态标志位，用于记录边的实时状态（如：可用、满载等） ，便于后

续根据状态进行针对性的控制

weight 实时权重，用于实现不同算法下的路由控制

基于上述建模方法，图 1 与图 2 所示的微模块机房空间布局可以转化为如下 带权无向图（图 3），机房中的各类桥架被抽象为图中的边，设备以及机柜映射 为节点，桥架的物理走向、空间关系、 长度、负载容量等参数则通过边的属性量 化表达。



**图** **3** **微模块平面布置抽象后的无向图**

**2.3 最短路由/成本优先策略的实现**

最短路由是最基础也是最常见的需求之一，其目标是找到两点之间物理距离 最短的路径。而成本优先本质上是加权的最短路由，可通过将物理距离映射为经 济成本来实现。

（1）算法的选择

机房建模完成后，线缆最短路由规划问题可抽象为图论中的最短路径搜索问 题。常见的寻路算法包括：

**表** **3** **最短路由算法对比**

|  |  |
| --- | --- |
| 算法 | 算法特点 |
| 广度优先搜索（BFS） Dijkstra 算法 | 仅适用于无权图或等权图，无法满足机房布线中多权重的需求  适用于单源最短路径搜索，能够处理非负权重图，但需遍历所有可能 路径，在大规模拓扑中效率较低 |
| A\*算法 | 结合启发式函数优化搜索方向，在已知目标节点时显著减少搜索范 围，兼具效率与准确性 |

考虑到数据中心场景下，图的规模可能较大（以常见的千台服务器的 POD 为例：总设备以及机柜数量在 1300-1500 左右，总线缆数量在一万条左右）， Dijkstra 算法的性能可能会成为瓶颈，而节点中记录的坐标信息，天然支持启发 式函数的构建，故本文选择 A\*算法作为最短路由方案的核心算法，当前场景下 边的权重 *w(e)*:

w(e) = Le

其中：*Le* *=base\_length×redundancy\_factor*，即实际的线缆长度。

（2）启发式函数的设计

启发式函数 *h(u,v)*是 A\*算法的核心优化机制，其设计需在搜索效率与路径 最优性间实现平衡，同时在三维机房布局中，启发式函数需准确反映节点间的空 间关系。典型候选方案包括：

**表** **4** **启发式函数对比**

|  |  |
| --- | --- |
| 启发式函数 | 特点 |
| 曼哈顿距离 | 适用于网格状路径约束场景，但忽略对角线方向的实际距离 |
| 切比雪夫距离 | 支持八方向移动，但对三维空间适应性不足 |
| 欧氏距离 | 直接计算三维空间直线距离，符合机房中线缆布线的物理特性 |

本方案采用三维欧氏距离作为启发式函数，并结合机房布局特点进行优化， 设计启发式函数 *h(u,v)*如下：



其中：

 *∆X、∆Y、∆Z* 为节点 *u* 与*v* 的坐标差值；

 *α、β、γ* 为为坐标转换系数，对应机房实际布局中*X*（水平）、*Y*（纵向）、 *Z*（垂直）方向的单位距离（毫米）；

 *C* 为成本单价函数，若未定义则退化为物理距离计算模式。

启发式函数 *h(u,v)*的可采纳性是确保路径最优性的核心条件，其数学定义为：

ℎ(u, v) ≤ ℎ\* (u, v)

其中 *h\** *(u,v)*表示起始节点 *u* 到目标节点 *v* 的实际最小代价。若 *h(u,v)*违反此 条件，算法可能因过度乐观的启发式估计而偏离全局最优路径。以图 1 、2 的情 况为例，一般情况下可按照 *α=600、β=2200、γ=5000* 来取值（即：标准机柜 宽度 600mm，厚度 1200mm+过道宽度 1000mm，楼层高度差 5000mm）。

（3）成本优先的实现

成本优先模式通过权重函数映射将物理距离转换为经济成本，具体实现可分 为静态与动态两种策略：

**表** **5** **成本优先模式的映射**

|  |  |
| --- | --- |
| 转换方案 | 说明 |
| 静态成本计算  动态成本函数 | 直接采用固定单价 *C*，定义边权重为：*w(e)=Le.C*  用户自定义成本函数 *w(e)=f(Le)*，实现例如：分段函数（实现类似区 间记价、或区分不同路径类型的单价）、非线性函数（引入规模效应， 类似线缆长度越长、单位成本越低）等功能。 |

**2.4 双路由策略的实现**

在安全等级要求较高的场景中（如政务、金融、核心网等），线缆双路由是 另一种常见的需求，其核心目标是为一组互联的设备提供两条完全独立的物理路 径，更大程度上确保不会出现单点故障。

该问题可抽象为求解图的 *k* 节点连通性（*k=2*），其本质是在图结构中寻找 两条不存在公共中间节点的路径，主要步骤如下：

（1）首次路径搜索

使用最短路由算法，计算源节点 *s* 到目标节点 *t* 的最优路径 *P1*；

（2）节点状态标记

将路径 *P1* 中所有中间节点（除 *s* 和 *t*），通过修改节点标志位（*flags*）的方 式，标记为“临时不可用”；

（3）二次路径搜索

在更新后的图 *GI*中重新执行路径搜索，生成与 *P1*节点不相交的次优路径 *P2*；

（4）完成双路径计算后，清除所有节点的临时标记，恢复节点原始状态以 保证后续路由计算的正确性。

**2.5 负载均衡策略的实现**

近年来数据中心单机架功率从 5kW 逐步提升至 7kW、10kW 甚至 20kW，装 机密度不断上升导致布线密度上升，为了避免局部线缆桥架负载过高，负载均衡 也逐渐变成了一种常见的需求。

负载均衡问题可抽象为带容量约束的图论优化问题：在图模型 *G=(V,E)*中， 每条边 *e∈E* 具有实时负载状态 *current\_load(e)*与容量上限 *max\_capacity(e)*，需 为每条线缆选择路径{*e1,e2,...,* *en*}，使得：

 路径总成本（如长度或者经济开销等）最小化；

 *∀ei* *∈path,* *current\_load(ei)≤max\_* *capacity(ei)*。

为实现上述目标，本文提出一种动态权重驱动的负载均衡算法，其核心是通 过实时感知桥架负载状态，调整路径权重以引导流量分布。

（1）动态权重函数的设计

定义边 *e* 的动态权重 *w(e)*为：

w(e) = Le . (1 + α . load\_factor . eβ.load\_factor ) 其中：

 *Le* *=base\_length×redundancy\_factor*，即实际的线缆长度；

  其中 *current\_load*与 *max\_capacity*分 别表示边的当前负载与最大容量；

 *α* 为全局敏感度系数，控制权重增长的总体强度；

 *β* 为非线性强度系数，调节高负载区域的指数放大效应。

该函数设计的目的是实现低负载时平缓增长、高负载时显著放大的非线性效

果，引导算法优先选择低负载路径。例如：

当 *load\_factor=0.3、α=2、β=3* 时：

w(e) = Le . (1 + 2 \* 0.3 \* e3\*0.3) ≈ 2.48Le

当 *load\_factor=0.8、α=2、β=3* 时：

w(e) = Le . (1 + 2 \* 0.8 \* e3\*0.8) ≈ 18.64Le

（2）敏感系数的作用机制

在动态权重函数 *w(e)*中，全局敏感度系数(*α*)和非线性强度系数(*β*)共 同决定了负载均衡策略的响应特性。

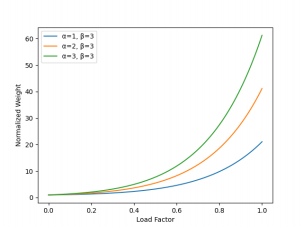
**表** **6** **敏感系数作用机制**

|  |  |
| --- | --- |
| 系数 | 说明 |
| 全局敏感度系数(*α*) | 整体控制权重增长的幅度。增大 *α* 会使所有负载水平的权重按 比例增加 |
| 非线性强度系数(*β*) | 决定高负载区域的权重陡增程度。增大 *β* 会使负载率接近 1 时 的权重急剧上升 |

二者的作用机制分析如下：

 **全局敏感度系数(*α*):**

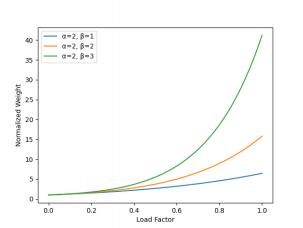
*α* 用来控制整体权重增长的幅度。如 4(a)所示，当 *β* 固定时，增大 *α* 会导致 所有负载水平下的权重按比例上升。例如，当 load\_factor=0.5 、 *β*=3 时， *α* 从 1 增至 3，权重从 1+1×0.5×e1.5 ≈3.24 倍提升至 1+3×0.5×e1.5 ≈7.72 倍。这表明， *α* 的调节能够全局增强系统对负载变化的敏感度。



**图** **4(a)** **α对权重的影响(β=3）**

 **非线性强度系数（*阝*）：**

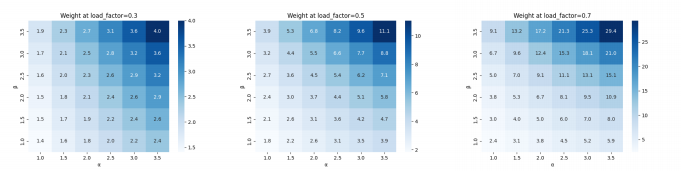
*阝* 决定高负载区域的权重陡增程度。如 4(b)所示，当 *α* 固定为 2 时，*阝* 的增 大会显著放大高负载（load\_factor>0.6）时的权重。例如， 在 load\_factor=0.8 时， *阝* 从 1 增至 3，权重从 1+2×0.8×e0.8 ≈4.56 倍跃升至 1+2×0.8×e2.4 ≈18.64 倍。 这种非线性特性使得算法能够优先规避接近满载的路径，从而有效均衡整体负载 分布。



**图** **4(b)** **β对权重的影响(α=2）**

 **敏感系数的协同：**

图 4(c)展示了 *α* 与 *阝* 的联合影响，当 *α* 与 *阝* 均较大时（比如均为 3.5），权 重在 load\_factor=0.7 时达到 29.4 倍，表明此时系统对高负载路径具有较强的规 避倾向。在实际工程中，建议根据实际需求，通过 Python 仿真确定最符合需求 的参数组合，比如：对于负载均衡需求较高的场景，选取较高的参数组合；而对 成本敏感的场景则可选择较低的参数组合，以平衡负载均衡效果与经济性。



**图** **4(c)** **α与β联合作用下的热力图（** **load\_factor=0.3/0.5/0.7）**

**3 算法实现及验证**

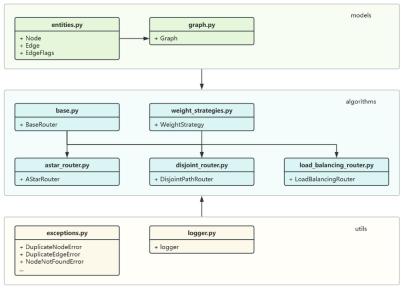
**3.1 实验环境介绍**

笔者使用 Python 3.10 对该算法进行了初步的实现，并使用一台2018 年上市 的老式笔记本电脑（配置：Intel® Core™ i5-8265U、8G DDR3 内存、256G SATA SSD、Window 10 LTSC 21H2 系统）进行测试，以观察该算法在较低的硬件条件 下的运行表现。

**3.2 项目关键实现**

（1）整体架构

考虑到后续代码的可扩展性和维护性，项目采用分层架构设计（如图 5）， 通过模块化实现算法、模型与工具的深度解耦。



**图** **5** **项目代码架构图**

具体层次说明如下：

**表** **7** **代码层次**

|  |  |
| --- | --- |
| 层次 | 说明 |
| 算法层 | 基于策略模式封装多种路由算法（A\*最短路由、双路由、动态负载 均衡等），通过继承 BaseRouter 基类实现统一接口，确保算法可插 拔性与扩展性 |
| 模型层 | 定义 Node 与 Edge 实体类，用来记录机房三维空间拓扑信息；Graph 类实现图结构的动态构建与缓存管理，支持桥架类型过滤与权重实时 |

计算

工具层 提供异常处理、日志记录等工具，保障系统鲁棒性

（2）可扩展性设计

**算法扩展：**新增路由算法仅需继承 BaseRouter 类并重写 find\_path 方法，例 如 LoadBalancingRouter 通过动态权重函数注入即可无缝接入现有框架。

# load\_balancing\_router.py

class LoadBalancingRouter(BaseRouter):

"""动态负载均衡路由器，计算路径后自动更新边负载"""

def \_\_init\_\_(self, graph: Graph, load\_sensitivity: float = 2.0):

# 使用负载均衡策略作为权重函数

weight\_func = lambda edge: WeightStrategy.load\_balanced(edge, load\_sensitivit y)

super().\_\_init\_\_(graph, weight\_func)

self.load\_sensitivity = load\_sensitivity

def find\_path(self, start\_node\_id: str, end\_node\_id: str, path\_type: str = "direc t", load\_increment: int = 1) -> List[Edge]:

# 检查节点是否存在

if not self.graph.get\_node\_by\_id(start\_node\_id):

raise NodeNotFoundError(f"Start node {start\_node\_id} not found") if not self.graph.get\_node\_by\_id(end\_node\_id):

raise NodeNotFoundError(f"End node {end\_node\_id} not found")

nx\_graph = self.graph.to\_networkx(path\_type, self.weight\_func)

try:

node\_path = nx.shortest\_path(nx\_graph, source=start\_node\_id, target=end\_n ode\_id, weight="weight")

except nx.NetworkXNoPath:

raise PathNotFoundError(f"No path from {start\_node\_id} to {end\_node\_id}")

# 提取边对象列表

edges = []

for u, v in zip(node\_path[:-1], node\_path[1:]):

edge\_data = nx\_graph.get\_edge\_data(u, v)

edge = next(iter(edge\_data.values()))["obj"]

edges.append(edge)

|  |
| --- |
| # 更新路径上的边负载 for edge in edges:  edge.current\_load += load\_increment  return edges |

**权重策略扩展**：WeightStrategy 类采用工厂模式，可通过添加静态方法实现 新型权重计算策略，实现集中管理策略逻辑，避免硬编码。

|  |
| --- |
| # weight\_strategies.py class WeightStrategy:  @staticmethod  def shortest\_length(edge: Edge) -> float:  """最短长度策略"""  return edge.effective\_length  @staticmethod  def cost\_priority(edge: Edge) -> float:  """成本优先策略"""  if edge.cost\_function:  return edge.cost\_function(edge.effective\_length)  return edge.effective\_length  @staticmethod  def load\_balanced(edge: Edge, alpha: float = 1.0, beta: float = 2.0) -> float: """负载均衡策略（支持动态敏感度）"""  base = edge.effective\_length  load\_factor = edge.load\_ratio  return base \* (1 + alpha \* load\_factor \* np.exp(beta \* load\_factor)) |

**状态机** ：Edge 类通过 flags 字段支持自定义状态标记，current\_load 与 max\_capacity 属性的动态更新机制可无缝适配未来新增的运维约束条件。

|  |
| --- |
| # entities.py  class Edge(object):  """边类"""  @property  def max\_capacity(*self*):  return *self*.\_max\_capacity  @max\_capacity.setter  def max\_capacity(*self*, *value*): |

|  |
| --- |
| if *value* < 0:  raise ValueError("最大负载不能为负数")  *self*.\_max\_capacity = *value*  @property  def current\_load(*self*):  return *self*.\_current\_load  @current\_load.setter  def current\_load(*self*, *value*):  if *value* < 0:  raise ValueError("负载不能为负数")  *self*.\_current\_load = *value*  *# 如果current\_load 超过max\_capacity，则设置F 标记*  *# current\_load 可以大于max\_capacity，设置* *F 标记但是不去掉* *U* if *self*.current\_load >= *self*.max\_capacity:  *self*.flags.set("F")  else:  *self*.flags.unset("F")  class EdgeFlags(object):  """Edge 的 Flags 类"""  FLAGS = {  'U': 1 << 0, # Edge 可用（up）  'F': 1 << 1, # Edge 满载（fully\_loaded，current\_load > max\_capacity） 'C': 1 << 2, # Edge 是夸楼层的（cross）  }  def set(self, char):  self.\_value |= self.FLAGS[char]  return self # 支持链式调用  def unset(self, char):  self.\_value &= ~self.FLAGS[char]  return self  def has(self, char):  return bool(self.\_value & self.FLAGS[char]) |

**多图策略**：机房中同一组起点与终点间可以存在多种不同类型的边（如：同 一组机柜间可能同时存在网线和光跳线），代码在设计时需引入多图策略，通过

*path\_type* 属性为不同类型的边定义不同的属性，路由计算时通过该属性约束边 的选择。

（3）动态权重重传机制

BaseRouter 通过 weight\_func 参数支持运行时权重策略注入。例如 ， LoadBalancingRouter 初始化时通过闭包捕获 load\_sensitivity 参数，动态生成权重 函数：

|  |
| --- |
| # base.py  class BaseRouter(ABC):  def \_\_init\_\_(self, graph: Graph, weight\_func: Callable[[Edge], float] = None):  self.graph = graph  self.weight\_func = weight\_func or self.\_default\_weight  # load\_balancing\_router.py  class LoadBalancingRouter(BaseRouter):  """动态负载均衡路由器，计算路径后自动更新边负载"""  def \_\_init\_\_(*self*, *graph*: Graph, *load\_sensitivity*: float = 2.0):  *# 使用负载均衡策略作为权重函数*  weight\_func = lambda *edge*: WeightStrategy.load\_balanced(*edge*, beta = *load\_sen sitivity*) |

（4）性能优化

在主体的机房建模以及路由计算等功能之外，对于代码效率以及性能的优化 也是必须考虑到的一环。一种比较常见的优化方式是使用缓存，比如：由于实际 需求，代码支持多种 path\_type 的边，比较好的做法是使用每种 path\_type 的边初 始化图后都进行缓存，以避免图的无效重复构建。

|  |
| --- |
| # graph.py  class Graph(object):  """图结构"""  def \_\_init\_\_(self):  ...  # 缓存 NetworkX 图对象  self.\_nx\_graph\_cache = {}  # 缓存有效性标志  self.\_cache\_valid = False  def add\_node(self, node: Node): |

|  |
| --- |
| ...  self.nodes[node.id] = node  self.invalidate\_cache()  def add\_edge(self, edge: Edge):  ...  self.edges[edge.id] = edge  self.invalidate\_cache()  def to\_networkx(self, path\_type: str = None, weight\_func: Callable[[Edge], float] = None) -> nx.MultiGraph:  """将当前图转换为 networkx 图对象，按 path\_type 自动过滤"""  # 生成缓存键（函数哈希+过滤条件）  cache\_key = (path\_type, hash(weight\_func.\_\_code\_\_)) if weight\_func else path\_ type  if cache\_key in self.\_nx\_graph\_cache and self.\_cache\_valid:  return self.\_nx\_graph\_cache[cache\_key]  ...  self.\_nx\_graph\_cache[cache\_key] = nx\_graph  self.\_cache\_valid = True  return nx\_graph  def invalidate\_cache(self):  """标记缓存失效（在边或节点变更时调用）"""  self.\_nx\_graph\_cache.clear()  self.\_cache\_valid = False |

另外一种比较常见的优化方式是使用多线程技术，使用线程池并发处理请求。 但是在使用时需确保部分属性（比如：current\_load 等）的修改是原子操作，或 者具备适当的锁机制。

**3.3 实验结果与分析**

本文选取了两种机房进行实验：双层结构的微模块机房和规模较大的单层机 房，微模块空间结构较为复杂，用来测试算法的适应性，单层大机房则用来测试 算法的性能。两种机房情况如下：

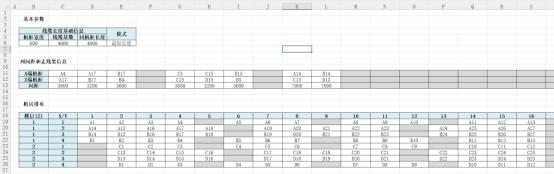
**表** **8** **测试机房情况介绍**

|  |  |
| --- | --- |
| 机房 | 说明 |
| 双层微模块 | 双层结构，每层有 4 列机柜，上下两层共 72 个机架。每层在 3 和 4 号机柜上方设置有走线桥架，在中间两列的末端（机房背面）设置有 垂直桥架连通上下两层 |
| 单层大机房 | 该机房共有 19 列 259 个机架，于列头、尾各设有主走线桥架，整个 机房共部署了约 1400 台服务器 |

（1）机房数据的输入

本文以某运营商已验收的工程线缆表为基准，在线缆表中通过辅助表的形式， 将所需的机房空间信息传递给测试脚本，测试脚本根据获得的基础数据，调用算 法计算出路由以及相应的线缆长度，最后写入表格。

输入的参数尽量同前期工程保持在同一量级，以减小由于算法不同（参数的 取定角度和方式不同）造成的差异，以便最终结果的比对，辅助表设计如下：



**图** **6** **测试用辅助表格式**

其中：

 机房排布：从机房的实际排布映射而来（参考图 1 、2）；

 列间距和走现架信息：记录了主走线架以及垂直走线架的位置、连接关 系以及距离信息；

 基本参数：一些基本参数， 同前期工程在计算时的数据尽量保持一致。

（2）适配性测试

为验证算法对复杂空间结构的适应性，本文选取双层微模块机房作为测试场 景，在测试中选取了 3 个部署设备类型不同的机房进行测试，共 2564 条数据，

通过对比新算法计算结果与基准数据的差异，得到如下统计结果：

**表** **9** **线缆长度对比统计**

|  |  |
| --- | --- |
| 指标 | 数值 |
| 样本数量 | 2564 |
| 完全一致比例( *Δ*=0） | 49% |
| 差异≤1米比例 | 68% |
| 差异≤2 米比例 | 90% |
| 平均差异（| *Δ*|） | -1.7 |
| 最大差异 | -3 |

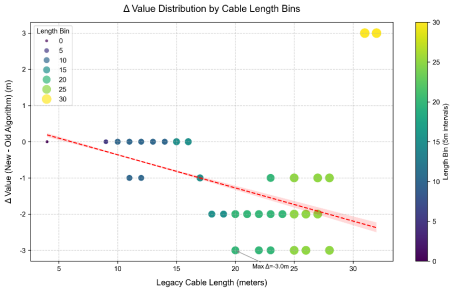
测试结果表明：新算法对复杂机房具有良好的适配性，90%的线缆长度差异 在 2 米以内，平均差异为-1.7 米，最大差异为-3米，属于合理范围之内。

对于线缆长度的差异，通过路径反查分析，差异主要源于两方面：

 颗粒度差异：新算法颗粒度提升明显，尤其在跨列、跨层的处理上有非 常大的不同，在较长的（跨列、跨层）线缆上会存在一定的差异；

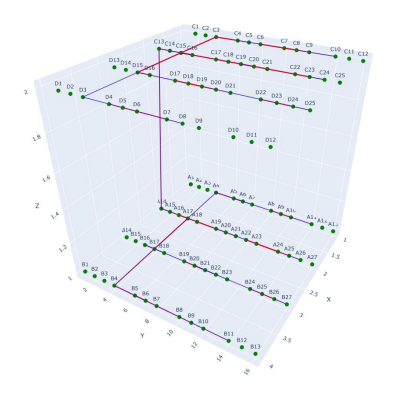
 冗余方式不同：为了控制变量，新算法在计算时冗余系数（ *redundanc* *y\_factor*）按照 1.0 来取值，而老方法在计算过程中还预留了一些冗余。

进一步以原始线缆长度为变量分析差异分布（图 7），发现长距离路由（>20 米）的差异更为显著，这与上述对长度差异的分析相符。



**图** **7** **线缆长度差异分析**

路径计算完毕后，利用算法中记录的信息，还可以尝试对机房排布以及走线 路由进行简单的可视化展示：



**图** **8** **微模块机房三维可视化**

（3）性能测试

本文选取一个单机房 259 机架的大型机房进行性能测试，测试用表共有 12 个 sheet 约 1200 行数据。经过多轮测试，性能数据如下：

**表** **10** **单层大机房测试性能数据**

|  |  |  |
| --- | --- | --- |
| 项目 | 平均用时（ms） | 说明 |
| 文件读取 | 2900 | 使用 openpyxl[5]载入线缆表 |
| 数据预处理 | 310 | 读取机房信息（辅助表），初始化图结构 |
| 路由计算 | 560 | 总时长，平均每行用时 0.5 ms 左右 |
| 文件保存 | 580 | 数据写入表格、保存 |

往期工程中，由于微模块复杂的空间结构，对于跨层的线缆只能手动筛选并 人工计算，或是在设备排布时进行特殊处理以规避该问题，增加了整体设计的难 度，同时在无形中也增加了出错的概率。

经测试本文的算法可以很好的适配空间结构复杂的机房，并在大规模计算中 也具有较好的性能。

**4 结论**

随着数据中心机房空间结构与布线复杂度的不断提升，传统路由规划方法在 精度、效率与策略多样性方面已难以满足工程需求。本文提出了一种基于三维图 建模的通用线缆路由算法，通过将机房空间结构抽象为带权无向图，结合 A\*算 法、不相交路径和动态权重等策略，实现了多种目标下的智能路由规划。本研究 的主要成果与价值体现在以下几方面：

（1）三维图建模方法

提出一种将机房物理空间抽象为带权无向图模型的方法，通过节点与边属性 的扩展，实现了对复杂空间关系的精准表征。实验表明， 该方法对空间结构较为 复杂的微模块机房（双层结构）适配良好，此外节点与边的属性设计具备高度可 扩展性，未来可通过新增属性字段适配更复杂的机房情况，为数据中心的智能化 规划提供了理论支撑。

（2）多元化策略实现

提出动态权重函数与负载敏感系数机制，利用节点状态以及边的动态属性， 针对目前工程中常见的需求，给出了如最短路由、成本优先、双路由、动态负载 均衡等模式的实现方法。并且就实现过程中的一些关键点（如算法扩展、策略扩 展、多图特性、性能优化等）给出了建议和示例。

（3）实际工程应用效果

以某运营商已验收的工程为基准，针对实际的工程文件以及机房情况，设计 了测试方案，验证了算法的可行性。此外还针对复杂空间结构和大批量计算两类 典型场景进行了测试，同时在现有方法的基础上进行了扩展性尝试（如简单的可 视化实现），实验证明该方法具有一定的实用价值。

目前本文算法仅针对一些常见的布线需求进行了实现，未来研究可从以下方 向展开：1、结合可视化功能，实现可视化施工指导、可视化运维（比如断点排 查）等；2、完善控制功能，实现类似 ACL、路由协议的功能， 实现更加灵活的 路由控制；3、结合机器学习技术，利用历史布线数据优化动态权重策略，进一 步提升负载策略的精度。

**参考文献**

[1] 钞小静. 新型数字基础设施促进我国高质量发展的路径[J]. 西安财经学院学报,2020,33 (2):15-19. DOI:10.3969/j.issn.1672-2817.2020.02.003.

[2] 中国移动通信集团设计院有限公司, 中国移动通信集团有限公司. 微模块:CN2022113908 39.1[P]. 2024-05-07.

[3] 综合布线系统工程设计规范:GB 50311-2016[S]. 2016.

[4] David Beazley, Brian K. Jones. Python Cookbook, 3rd Edition[M]. 人民邮电出版社, 2 015 年.

[5] Eric Gazoni, Charlie Clark. openpyxl - A Python library to read/write Excel 2010 xlsx /xlsm [EB/OL]. <https://openpyxl.readthedocs.io/en/stable>.