# 综合实验报告

## 广州地铁线路规划系统 Guangzhou Metro Query System

## 一、作者

华南师范大学
软件工程
2022 级 黄子健

## 二、问题描述和基本要求

### 2.1 问题描述

基于广州地铁系统，设计一个系统终端来进行地铁线路规划与信息查询。用户希望系统提供最短路线的计算与推荐功能，例如路线最短、时间最短、换乘最少等信息，并提供适合用户操作的良好的界面（UI）来展示这些信息。考虑到简化问题，本问题仅考虑广州地铁的 1 号线、2 号线和 3 号线，忽略了其它线路。地铁站点之间的路线长度只需合理估算。为了实现这一功能，可将广州地铁路线图建模为带权无向图，其中顶点表示站点，边表示站点之间的道路，边上的权值表示距离。

### 2.2 基本要求

1. 选择合适的数据结构将广州地铁路线信息存入文件，使系统能自动读取并建图。
2. 实现功能：输入两个站点名称后能够得到它们之间的最短线路与最少站点路径，如果两者不可达，则给出相应提示。
3. 在展示线路时，显示路径的长度（或时间）。
4. UI 界面美观简洁，方便用户操作。

## 三、工具/准备工作

### 3.1 涉及的数据结构与算法知识

1. 图论（图的存储（邻接表）、单源最短路算法（Dijkstra））
2. 优先队列 PriorityQueue

### 3.2 开发工具

运行环境要求：Windows

开发语言：Python 3.10

集成开发环境：PyCharm

Dependency：networkx、tkinter、PIL、matplotlib

## 4、 分析与实现

### 4.1 地铁路线图的存储

为了实现地铁线路的高效存储与系统的自动读取，需要找到一种合适的数据结构。本项目选择了图论中的**图（Graph）。**具体来说，采用了**邻接表（Adjacency List）**的方式来存储广州地铁路线信息，其中顶点表示地铁站点，边表示站点之间的连接关系，边上的权值表示距离。由于本项目只考虑一二三号线，地铁网络的图形可能很稀疏，因此与邻接矩阵相比，使用邻接表可降低项目的内存需求。

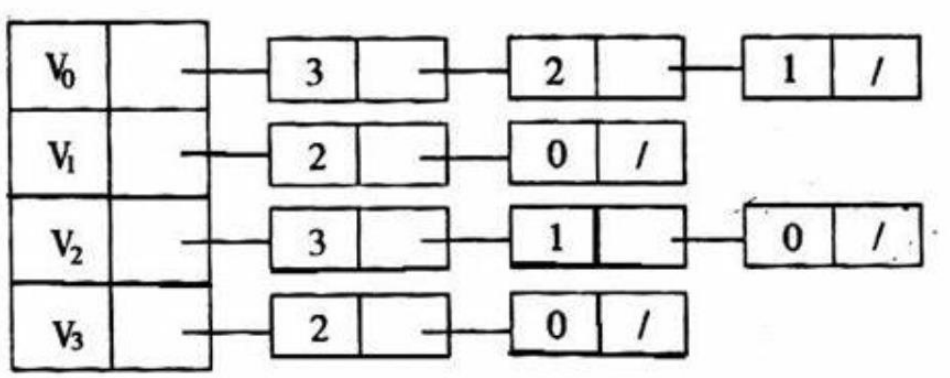



图 邻接表 Adjacency List

本项目使用 subway_data.txt 文件来存储广州地铁一二三号线的邻接表，文件的数据格式为每一行包含三个值，用逗号分隔，表示两个地铁站之间的连接关系和距离：

```
西朗，坑口，1.3
坑口，花地湾，0.9
花地湾，芳村，1.2
```

这样的数据表示西朗和坑口之间有一条地铁线路，距离为 1.3km，坑口和花地湾之间有一条地铁线路，距离为 0.9km，依此类推。这种格式的数据能够被项目代码中的 read_subway_data 函数正确解析并构建成一个图，该函数涉及到 python 的 networkx 库，用于创建和操作图形：

```python
# 读取地铁路线图信息，构建图
def read_subway_data(file_path):
    subway_graph = nx.Graph()
    with open(file_path, 'r', encoding='utf-8') as file:
        for line in file:
            data = line.strip().split(', ')
            station1, station2, distance = data[0], data[1], float(data[2])
            subway_graph.add_edge(station1, station2, distance=distance)
    return subway_graph
```

### 4.2 路径规划算法的实现

根据要求，对于输入的两个站点，本项目需要实现两种路径规划算法：最短路径方案、最少站点方案。其中，最短路径方案即从起始站点到目标站点的路线中寻找一条总公里数最少的路径方案，最少站点方案即寻找一条途径站点数最少的路径方案。

对于最短路径方案，实质上是个单源最短路径问题。考虑到地铁站点间的距离不可能为 0 或负数，可以使用 **Dijkstra 算法**，通过在图中逐步扩展最短路径的方式找到目标站点。同时，可以使用数据结构**优先队列 PriorityQueue**（即最小堆）来优化 Dijkstra。由于最小堆能够在常数时间内找到具有最小优先级的元素，因此可以高效选择当前最短路径距离的节点。代码使用 python 的 heapq 模块来实现优先队列。

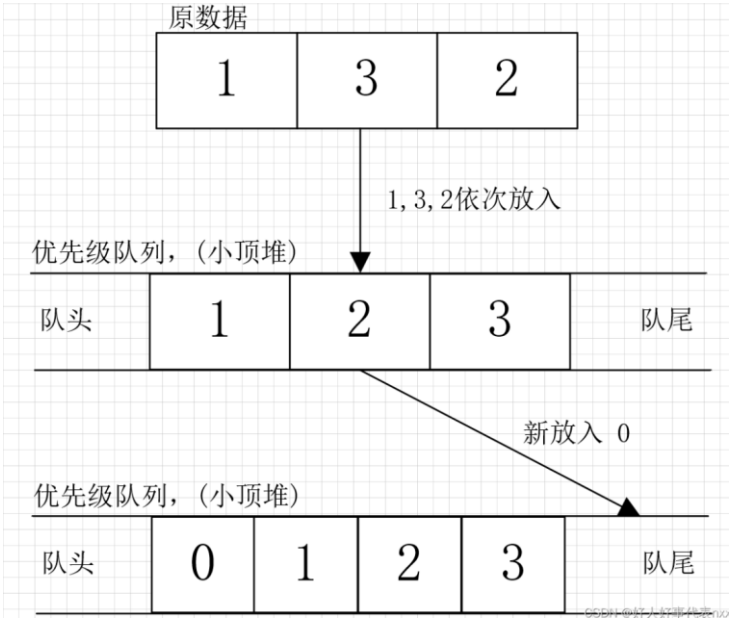



图 优先级队列（最小堆）

以下是 Dijkstra 算法的主要步骤：

1. **初始化：** 将起始节点的最短路径距离设置为 0，将其他节点的最短路径距离设置为无穷大。将所有节点标记为未访问。将所有节点加入优先队列中，以节点的最短路径距离作为优先级。
2. **选择最短路径节点：** 从未访问的节点中选择具有最小最短路径距离的节点。
3. **更新最短路径距离：** 对于选定的节点，更新与其相邻节点的最短路径距离。如果通过当前节点到达相邻节点的路径比已知的最短路径短，更新最短路径，并将相邻节点重新加入优先队列。
4. **标记节点为已访问：** 将选定的节点标记为已访问，表示已经找到从起始节点到该节点的最短路径。
5. **重复步骤 2-4：** 重复选择最短路径节点、更新距离和标记节点的步骤，直到目标节点被标记为已访问或所有可达节点都被标记为已访问。
6. **路径重构：** 从目标节点回溯到起始节点，重构最短路径。

```
# dijkstra：计算两站点间最短路径,单源最短路算法
def dijkstra(subway_graph, start_station, end_station):
    # 初始化所有节点的距离为无穷大
    distances = {node: float('infinity') for node in subway_graph.nodes}
    distances[start_station] = 0  # 从起始到起始站的距离为 0
    predecessors = {node: None for node in subway_graph.nodes}
    # 使用优先队列，用于跟踪节点和它们当前的距离
    priority_queue = [(0, start_station)]
    # Dijkstra 算法步骤
    while priority_queue:
        # 已知距离最小的节点
        current_distance, current_station = heapq.heappop(priority_queue)
        # 如果当前距离大于已知距离，则跳过
        if current_distance > distances[current_station]:
            continue
        # 探索当前节点的邻节点
        for neighbor, data in subway_graph[current_station].items():
            weight = data['distance']
            # 计算当前节点到邻节点的总距离
            distance = current_distance + weight
            # 如果找到更短的路径，更新距离和前驱节点
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                predecessors[neighbor] = current_station
```

```
                heapq.heappush(priority_queue, (distance, neighbor))
    # 从终点到起点重建路径
    path = []
    current_station = end_station
    while current_station is not None:
        path.insert(0, current_station)
        current_station = predecessors[current_station]
    # 检查是否存在从起点到终点的有效路径
    if path[0] == start_station:
        return path, distances[end_station]  # 返回路径和总距离
    else: # 未找到有效路径
        return None, None
```

对于最少站点方案，即：给定两站点，寻找一条途径站点数最少的路径方案。由于该方案并不需要考虑站点间的路径距离，因此可以考虑将站点间距离均设为 1。可以发现，当距离均为 1 时，两站点之间的路径距离即它们之间的站点数 -1。因此可以将最少站点路径转换为：在距离均为 1 的地铁图中寻找最短路径，其中最少站点数量即最短路径的距离 +1。同样使用 Dijkstra 算法，只需要多加一步初始化边权为 1：

```
# dijkstra：把边权设置为 1，两点间最短路径即途径站点数最少的路径
def dijkstra_min_node(subway_graph, start_station, end_station):
    # 把边权设置为 1
    for u, v in subway_graph.edges():
        subway_graph[u][v]['distance'] = 1
    ......
```

### 4.3. UI 界面的设计

为了提供简单直观的用户交互界面，本项目选择使用 Python 内建的 GUI 库 tkinter。

使用 tkinter 创建系统主窗口，包含输入框、按钮等控件。用户通过这些控件与系统进行交互，输入起始站点和目标站点，然后点击按钮进行方案查询。查询按钮触发后，调用相应的函数执行路径查询。

代码如下：

```
# UI 界面
class SubwayUI(tk.Tk):
    def __init__(self, subway_graph):
        super().__init__()
        self.title("广州地铁路线查询系统")
```

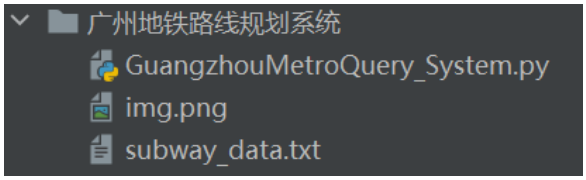
```
        self.geometry("700x500")
        self.subway_graph = subway_graph
        # 输入框
        self.label1 = tk.Label(self, text="起始站点:")
        self.entry1 = tk.Entry(self)
        ......
          # 查询按钮
        self.button = tk.Button(self, text="查询路径方案
",command=self.query_path)
        ......
         # 图片
        original_image = Image.open("img.png")
        ......
        # 消息提示
        self.label3 = tk.Label(self, text="目前仅支持一号线、二号线、三号线的
查询。")
        ......
        # 添加垂直分割线
        self.separator = ttk.Separator(self, orient="vertical")
        ......
        # 划分右半区
        self.right_frame = tk.Frame(self, bg="lightgray")
        self.right_frame.place(x=351, y=0, relwidth=1, relheight=1)
        ......
        # 输出结果部分
        self.result_empty1 = tk.Message(self.right_frame, text="",
width=300, bg="lightgray")
        self.result_empty1.pack(expand=False, side="top", anchor="w")
        ......
        # 退出按钮
        self.button = tk.Button(self, text="退出", command=self.exit_system)
        self.button.place(x=630, y=450, width=50, height=36)

    # 查询按钮事件
    def query_path(self):
        ......
```

```
    # 退出按钮事件
    def exit_system(self):
        ......
```

UI 界面如下：


## 五、测试与结论

**项目结构：**




其中，GuangzhouMetroQuery_System 为项目代码，img.png 为系统展示的广州地铁图，subway_data.txt 为存储的一二三号线地图数据。

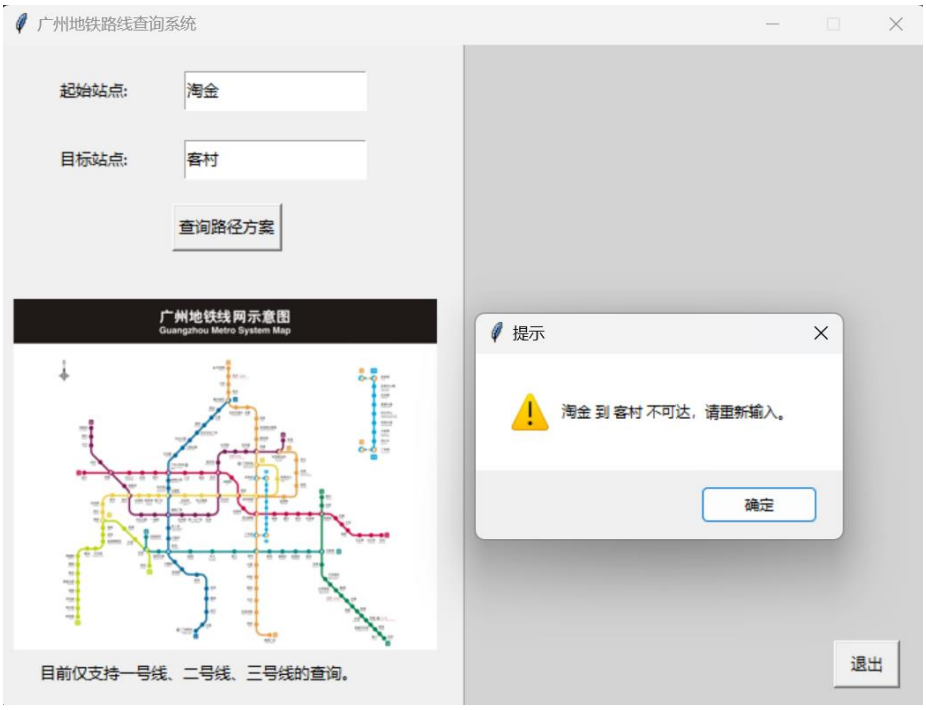
**测试一：**

输入**可联通**的两个站点，以客村与广州南为例：


由于只考虑一二三号线，因此可选择的路径并不多。在该例中，两条规划线路为同样的线路是正常现象。

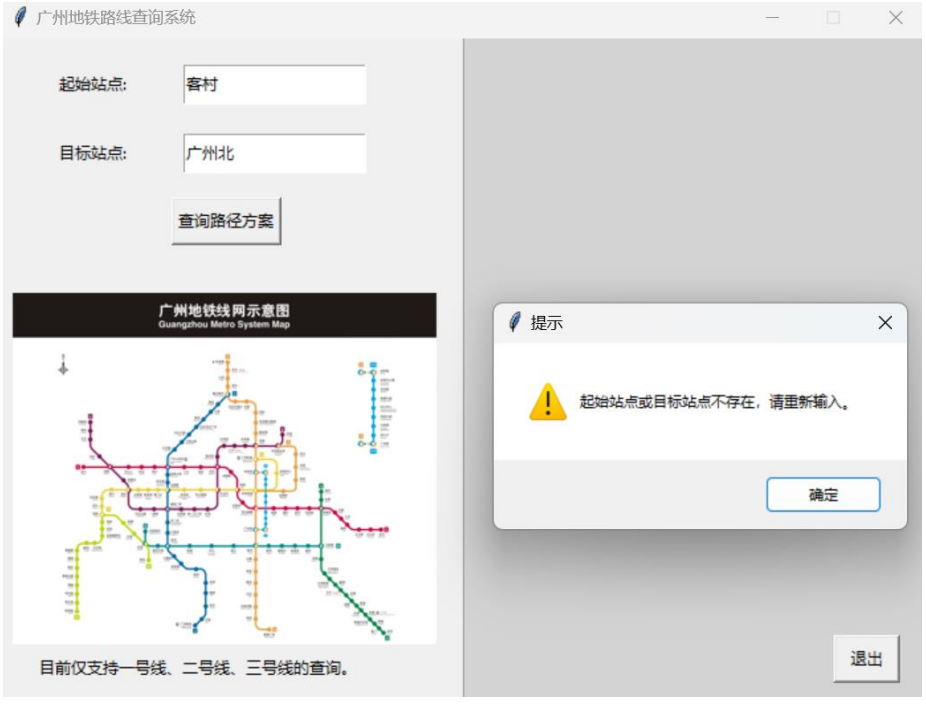
**测试二：**

输入**不可联通**的两个站点，以客村与淘金为例。由于淘金为五号线，本系统只考虑一二三号线，因此三号线的客村与淘金不可能联通。为了测试，将淘金→区庄的路径输入到地图集中，以保证淘金站点为图中存在的节点：




**测试三：**

输入**不存在**的站点，以客村与广州北（虚构站点）为例。



**测试四：**

测试退出按钮的功能。


## 六、综合实验总结

### 6.1 项目综述

在本项目中，采用了图论中的图（Graph）作为地铁线路的高效存储方式，选择了邻接表（Adjacency List）来表示广州地铁一二三号线的路线信息。这种数据结构通过顶点表示地铁站点、边表示站点之间的连接关系，且边上的权值表示距离，有效降低了内存需求，尤其适用于稀疏的地铁网络。通过 subway_data.txt 文件存储地铁邻接表，该文件的格式清晰且易于解析，方便项目代码中的 read_subway_data 函数将地铁线路信息动态构建成图，展现了较高的灵活性和可扩展性。

在路径规划算法的实现方面，本项目着重考虑了两种方案：最短路径方案和最少站点方案。通过合理的算法选择和数据结构设计，成功将两种问题都转化为单源最短路径问题，采用了 Dijkstra 算法，结合优先队列（最小堆）来提高运行效率。算法的步骤清晰，包括初始化、选择最短路径节点、更新最短路径距离、标记节点为已访问、重复步骤、路径重构等，确保了准确且高效的路径查询。

在功能扩展方面，项目囊括了广州地铁一二三号线的详细信息，并通过用户友好的 UI 界面使用 tkinter 库，提供了便捷的交互方式。用户可以通过输入起始站点和目标站点，点击按钮进行路径查询，使得系统更易用、直观。

### 6.2 创新点

本项目创新的部分主要体现在对路径规划算法的转化求解、Dijkstra 算法的优化结合运用、UI 界面的设计，以及数据结构的合理选择。通过这些创新，项目在路径规划和用户交互方面实现了较为出色的效果。

对于《数据结构与算法》课程而言，本项目最有价值的部分在于对数据结构与算法在实际场景中的运用。通过处理实际数据，解决了地铁路径规划问题，使得理论知识更具实际应用意义。通过这个项目，我更深刻地理解了数据结构、算法的重要性，同时也培养了解决实际问题的能力。同时，UI 界面的设计使得算法成果更直观地展现给用户，增加了项目的丰富性、美观性、实用性。

### 6.3 项目改进方向

在进一步的学习中，可以考虑深入研究其他路径规划算法（考虑 Floyd-Warshall 算法、A*算法等），以及更加复杂的地铁网络情景（考虑整个广州地铁线路）。此外，对于 UI 界面的设计，可以进一步优化，增加更多用户友好的功能和交互方式，可以考虑设计成小程序或网页模式。