【华为OD机考 统一考试机试C卷】快递员的烦恼（C++ Java JavaScript Python）

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷

2023年11月份，华为官方已经将 华为OD机考：OD统一考试（A卷 / B卷）切换到 OD统一考试（C卷）和 OD统一考试（D卷） 。根据考友反馈：目前抽到的试卷为B卷或C卷/D卷，其中C卷居多 ，按照之前的经验C卷D卷部分考题会复用A卷/B卷题，博主正积极从考过的同学收集C卷和D卷真题，可以查看下面的真题目录。

真题目录：华为OD机考机试 真题目录（C卷 + D卷 + B卷 + A卷） + 考点说明
专栏：2023华为OD机试(B卷+C卷+D卷)（C++JavaJSPy）
华为OD面试真题精选：华为OD面试真题精选
在线OJ：点击立即刷题，模拟真实机考环境 华为OD机考B卷C卷华为OD机考华为OD机考B卷华为OD机试B卷华为OD机试C卷华为OD机考C卷华为OD机考D卷题目华为OD机考C卷/D卷答案华为OD机考C卷/D卷解析华为OD机考C卷和D卷真题华为OD机考C卷和D卷题解

题目描述

快递公司每日早晨，给每位快递员推送需要送到客户手中的快递以及路线信息，快递员自己又查找了一些客户与客户之间的路线距离信息，请你依据这些信息，给快递员设计一条最例短路径， 告诉他最短路径的距离。

注意：

1. 不限制快递包裹送到客户手中的顺序，但必须保证都送到客户手中
2. 用例保证一定存在投递站到每位客户之间的路线，但不保证客户与客户之间有路线，客户位置及投递站均允许多次经过。
3. 所有快递送完后，快递员需回到投递站

输入描述

首行输入两个正整数n,m。

接下面n行，输入快递公司发布的客户快递信息，
格式为：客户id 投递站到客户之间的距离distance

再接下来的m行，是快递员自行查找的客户与客户之间的距离信息，
格式为：客户1id 客户2id distance

在每行数据中，数据与数据之间均以单个空格分割

规格;
0 < n <=10
0 <= m <= 10
0< 客户id <= 1000
0< distance <= 10000

输出描述

最短路径距离，如无法找到，请输出 -1

用例1

输入

```
2 1
1 1000
2 1200
1 2 300
```

输出

```
2500
```

说明

> 路径1： 快递员先把快递送到客户1手中，接下来直接走客户1到客户2之间的直通路线，最后走投递站与客户2之间的路，回到投递站，距离为1000 + 300 + 1200 = 2500
> 路径2： 快递员先把快递送到客户1手中，接下来回快递站，再出发把客户2的快递送到，在返回到快递站，距离为：1000 + 1000 + 1200 + 1200 = 4400
> 路径3：快递员先把快递送到客户2手中，接下来直接走客户2到客户1之间的直通线路，最后走投递站和客户1之间的路，回到投递站，距离为1200 + 300 + 1000 =2500
> 其他路径……
> 所有路径中，最短路径距离为2500

用例2

输入

```
5  1
5 1000
9 1200
17 300
132 700
500 2300
5 9 400
```

输出

```
9200
```

> 说明：在所有可行的路径中，最短路径长度为1000 + 400 + 1200 + 300 + 300 + 700 + 700 + 2300 + 2300 = 9200

解题思路

- 读取每个客户的ID和该客户到配送站的距离，更新距离矩阵和映射字典。
- 读取额外路线信息，更新距离矩阵中客户之间的距离。
- 创建一个动态规划数组，用于记录到达每个状态（客户的访问情况）的最短距离。
- 初始化一个队列，用于执行广度优先搜索（BFS）。

1. **广度优先搜索和动态规划**：
   - 从配送站开始，对每个可能的客户位置进行搜索。
   - 对于当前位置，遍历所有其他客户位置，检查是否存在一条到达下一个客户的路线。
   - 计算到达下一个客户的新状态，如果该状态未被访问过或者可以通过更短的距离到达，则更新动态规划数组。
   - 将新状态加入队列，以便继续搜索。
2. **搜索结束条件**：
   - 当队列为空时，搜索结束。
3. **输出结果**：

- 从动态规划数组中获取访问所有客户并返回配送站的最短距离。
- 如果最短距离不是无穷大，则存在有效路径，输出最短距离；否则输出-1表示无法访问所有客户。

在这个过程中，**动态规划**用于存储和更新到达每个客户集合状态的最短距离，而**广度优先搜索**用于遍历所有可能的客户访问顺序。动态规划数组的索引表示客户访问的状态（使用位掩码表示哪些客户已被访问），值表示到达该状态的最短距离。通过结合这两种方法，算法能够有效地找到访问所有客户并返回配送站的最短路径。

扩展：

位掩码（Bitmask）是一种利用位操作（比如位与、位或、位非等）来处理数据的技术。在编程中，位掩码通常用于以下目的：

1. **表示状态集合**：
   - 通过单个整数的不同位来表示多个状态或选项。例如，如果有 4 个选项，可以用一个 4 位的数字来表示这些选项的开启或关闭状态（0 表示关闭，1 表示开启）。如 `1010` 可以表示第一个和第三个选项被选中。
2. **高效的状态操作**：
   - 位掩码允许使用位操作来高效地改变、查询或比较状态。例如，使用位与操作（`&`）检查特定位的状态，使用位或操作（`|`）设置状态，使用位异或操作（`^`）切换状态。
3. **节省空间**：
   - 相比于使用多个布尔变量或者数组，位掩码通过将多个状态压缩在一个整数内，能更加节省空间。

在本题中位掩码用于表示不同客户的访问状态。每个位对应一个客户，如果该位为1，则表示相应的客户已被访问；如果为0，则表示客户未被访问。

模拟计算过程

对于给定的用例，我们有2个客户和1条额外的路线。下面是模拟计算过程的详细步骤：

1. **初始化变量**：
   - 客户数量 `n = 2`，额外路线数量 `m = 1`。
   - 无穷大值 `inf` 用于初始化距离。
   - 距离矩阵 `dis` 初始化为 `[[inf, inf, inf], [inf, inf, inf], [inf, inf, inf]]`。
   - 映射字典 `mapping` 初始化为空。
2. **构建距离矩阵和映射字典**：

- 客户1的ID为1，到配送站的距离为1000，更新 `dis` 和 `mapping`： `dis[0][1] = dis[1][0] = 1000`， `mapping[1] = 1`。
- 客户2的ID为2，到配送站的距离为1200，更新 `dis` 和 `mapping`： `dis[0][2] = dis[2][0] = 1200`， `mapping[2] = 2`。
- 额外路线连接客户1和客户2，距离为300，更新 `dis`： `dis[1][2] = dis[2][1] = 300`。

3. **初始化动态规划数组和队列**：

   - 动态规划数组 `f` 初始化为 `[[inf, inf, inf], [inf, inf, inf], [inf, inf, inf], [inf, inf, inf]]`。
   - 队列 `q` 初始化为 `[(0, 0, 0)]`。

4. **广度优先搜索**：

   - 从队列中取出 `(0, 0, 0)`，表示当前没有访问任何客户，位置在配送站，当前距离为0。
   - 尝试访问客户1，状态更新为 `1 << (1 - 1) = 1`，距离更新为 `0 + 1000 = 1000`，将 `(1, 1, 1000)` 加入队列。
   - 尝试访问客户2，状态更新为 `1 << (2 - 1) = 2`，距离更新为 `0 + 1200 = 1200`，将 `(2, 2, 1200)` 加入队列。
   - 队列现在为 `[(1, 1, 1000), (2, 2, 1200)]`。

5. **继续广度优先搜索**：

   - 取出 `(1, 1, 1000)`，尝试访问客户2，状态更新为 `1 | (1 << (2 - 1)) = 3`，距离更新为 `1000 + 300 = 1300`，将 `(3, 2, 1300)` 加入队列。
   - 取出 `(2, 2, 1200)`，尝试访问客户1，状态更新为 `2 | (1 << (1 - 1)) = 3`，距离更新为 `1200 + 300 = 1500`，但由于状态 `3` 已经以更短的距离 `1300` 被访问过，所以不更新。

6. **输出结果**：

   - 最终状态为 `3`，表示所有客户都被访问过，返回配送站的距离为 `f[3][0]`。
   - 由于我们没有直接从客户返回配送站的距离，我们需要考虑从最后一个客户返回配送站的距离。
   - 最短路径为从配送站到客户1，然后到客户2，再返回配送站，距离为 `1000 + 300 + 1200 = 2500`。
   - 输出最短距离 `2500`。

C++

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
```

```cpp
#include <limits>

using namespace std;

int main() {
    int n, r;
    cin >> n >> r; // 读取客户数量和路线数量

    // 定义无穷大用于初始化距离矩阵
    const int inf = numeric_limits<int>::max();

    // 初始化距离矩阵，所有值设为无穷大
    vector<vector<int>> d(n + 1, vector<int>(n + 1, inf));

    // 创建映射字典，将客户ID映射到矩阵索引
    unordered_map<int, int> m;

    // 读取客户信息，填充距离矩阵和映射字典
    for (int i = 0; i < n; ++i) {
        int id, dist;
        cin >> id >> dist;
        m[id] = i + 1;
        d[0][i + 1] = d[i + 1][0] = dist;
    }

    // 读取额外的路线信息，更新距离矩阵
    for (int i = 0; i < r; ++i) {
        int id1, id2, dist;
        cin >> id1 >> id2 >> dist;
        int i1 = m[id1];
        int i2 = m[id2];
        d[i1][i2] = d[i2][i1] = dist;
    }

    // 初始化动态规划数组，用于记录到达每个状态的最短距离
    vector<vector<int>> dp(1 << n, vector<int>(n + 1, inf));

    // 初始化队列，用于广度优先搜索
    queue<vector<int>> q;
    q.push({0, 0, 0}); // 将起始状态加入队列

```

```
    // 设置起始点到起始点的距离为0
    dp[0][0] = 0;

    // 执行广度优先搜索
    while (!q.empty()) {
        vector<int> curr = q.front();
        q.pop();
        int cs = curr[0], cp = curr[1]; // 当前状态和当前位置
        int cd = dp[cs][cp]; // 当前状态的最短距离
        for (int np = 0; np <= n; ++np) { // 遍历所有可能的下一个位置
            if (np != cp && d[cp][np] != inf) { // 如果下一个位置不是当前位置且可达
                int ns = (np != 0) ? cs | (1 << (np - 1)) : cs; // 计算新状态
                if (dp[ns][np] > cd + d[cp][np]) { // 如果新状态的距离可以被更新
                    dp[ns][np] = cd + d[cp][np]; // 更新距离
                    q.push({ns, np, 0}); // 将新状态加入队列
                }
            }
        }
    }

    // 获取访问所有客户并返回配送站的最短距离
    int min_d = dp[(1 << n) - 1][0];

    // 打印最短距离或-1（如果不存在有效路径）
    cout << (min_d != inf ? min_d : -1) << endl;

    return 0;
}
```

Java

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Map;
import java.util.Queue;
import java.util.Scanner;

public class Main {
```

```java
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    // 读取客户数量和路线数量
    int n = scanner.nextInt();
    int r = scanner.nextInt();

    // 定义无穷大用于初始化距离矩阵
    int inf = Integer.MAX_VALUE;

    // 初始化距离矩阵，所有值设为无穷大
    int[][] d = new int[n + 1][n + 1];
    for (int[] row : d) {
        Arrays.fill(row, inf);
    }

    // 创建映射字典，将客户ID映射到矩阵索引
    Map<Integer, Integer> m = new HashMap<>();

    // 读取客户信息，填充距离矩阵和映射字典
    for (int i = 0; i < n; i++) {
        int id = scanner.nextInt();
        int dist = scanner.nextInt();
        m.put(id, i + 1);
        d[0][i + 1] = d[i + 1][0] = dist;
    }

    // 读取额外的路线信息，更新距离矩阵
    for (int i = 0; i < r; i++) {
        int id1 = scanner.nextInt();
        int id2 = scanner.nextInt();
        int dist = scanner.nextInt();
        int i1 = m.get(id1);
        int i2 = m.get(id2);
        d[i1][i2] = d[i2][i1] = dist;
    }

    // 初始化动态规划数组，用于记录到达每个状态的最短距离
    int[][] dp = new int[1 << n][n + 1];
    for (int[] row : dp) {
```

```
            Arrays.fill(row, inf);
        }

        // 初始化队列，用于广度优先搜索
        Queue<int[]> q = new LinkedList<>();
        q.add(new int[]{0, 0, 0});

        // 设置起始点到起始点的距离为0
        dp[0][0] = 0;

        // 执行广度优先搜索
        while (!q.isEmpty()) {
            int[] curr = q.poll();
            int cs = curr[0], cp = curr[1];
            int cd = dp[cs][cp];
            for (int np = 0; np <= n; np++) {
                if (np != cp && d[cp][np] != inf) {
                    int ns = (np != 0) ? cs | (1 << (np - 1)) : cs;
                    if (dp[ns][np] > cd + d[cp][np]) {
                        dp[ns][np] = cd + d[cp][np];
                        q.add(new int[]{ns, np, 0});
                    }
                }
            }
        }

        // 获取访问所有客户并返回配送站的最短距离
        int min_d = dp[(1 << n) - 1][0];

        // 打印最短距离或-1（如果不存在有效路径）
        System.out.println(min_d != inf ? min_d : -1);
    }
}
```

javaScript

```
1
```

Python

```python
# 读取客户数量和路线数量
n, r = map(int, input().split())

# 定义无穷大用于初始化距离矩阵
inf = float('inf')

# 初始化距离矩阵，所有值设为无穷大
d = [[inf] * (n + 1) for _ in range(n + 1)]

# 创建映射字典，将客户ID映射到矩阵索引
m = {}

# 读取客户信息，填充距离矩阵和映射字典
for i in range(n):
    id, dist = map(int, input().split())  # 读取客户ID和到配送站的距离
    m[id] = i + 1  # 将客户ID映射到索引
    d[0][i + 1] = d[i + 1][0] = dist  # 设置配送站到客户和客户到配送站的距离

# 读取额外的路线信息，更新距离矩阵
for _ in range(r):
    id1, id2, dist = map(int, input().split())  # 读取两个客户ID和它们之间的距离
    i1 = m[id1]  # 获取客户1的矩阵索引
    i2 = m[id2]  # 获取客户2的矩阵索引
    d[i1][i2] = d[i2][i1] = dist  # 设置客户1到客户2和客户2到客户1的距离

# 初始化动态规划数组，用于记录到达每个状态的最短距离
dp = [[inf] * (n + 1) for _ in range(1 << n)]

# 初始化队列，用于广度优先搜索
q = [(0, 0, 0)]

# 设置起始点到起始点的距离为0
dp[0][0] = 0

# 执行广度优先搜索
while q:
    cs, cp, cd = q.pop(0)  # 取出当前状态，当前位置和当前距离
    for np in range(n + 1):
        # 如果下一个位置不是当前位置且两者之间有路线
        if np != cp and d[cp][np] != inf:
```

```
            # 计算下一个状态，如果不是配送站，则更新状态
            ns = cs | (1 << (np - 1)) if np != 0 else cs
            # 如果到达下一个状态的距离更短，则更新动态规划数组并将其加入队列
            if dp[ns][np] > cd + d[cp][np]:
                dp[ns][np] = cd + d[cp][np]
                q.append((ns, np, dp[ns][np]))

# 获取访问所有客户并返回配送站的最短距离
min_d = dp[-1][0]

# 打印最短距离或-1（如果不存在有效路径）
print(min_d if min_d != inf else -1)  # 打印结果
```

文章目录


机考真题
华为OD
No No No
CSDN @算法大师