第 1 章 数据关联的具体实现

数据关联的问题可以描述为：输入：当前时刻的检测结果（目标的 bbox）和之前存储的轨迹目标：找出前后两个时刻的相同目标，将他们进行关联。

问题 1：如何描述两个目标是相同的

相似度函数: 设 $S(\mathbf{x}, \mathbf{y})$ 是目标候选区域 **x** 与前一帧中的目标区域 **y** 之间的总相似度。该相似度函数可以表示为多个部分的加权和。

$$S(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} \omega_i \cdot S_i(\mathbf{x}, \mathbf{y})$$

其中，相似度函数的组成部分可能包括以下几种：

- 基于外貌特征，一般通过一个神经网络来进行提取 RE-ID(DeepSORT)。
- 表示运动学特征，一般用 IoU 来表示 (SORT)。
- 基于形状特征。

问题 2：数据关联问题重述

这里重述是为了区别于分配问题。

1.1 枚举法求解

1.2 贪婪算法求解

贪婪算法的求解速度显著高于其它算法，虽然其求得可能是局部最优解，不过由于任务的特殊性使得性能得到了保障。

1.3 SORT 的实现

SORT 论文中说到采用的是匈牙利算法。实际上，这是将数据关联问题转换成了一个分配问题，进而可以使用匈牙利算法进行求解。分配问题也可以用一个优化问题来进行描述：

对于 n×n 的代价矩阵 C，需要求得最小的总代价。

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

并且满足以下约束条件：

- 每个人只能完成一个任务且必须做一个任务：

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j = 1, 2, \ldots, n$$

- 每个任务只能由一个人完成且必须完成：

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i = 1, 2, \ldots, n$$

- x_{ij} 是二进制变量：

$$x_{ij} \in \{0, 1\} \quad \forall i, j = 1, 2, \ldots, n$$

其中 c_{ij} 表示第 i 个任务由第 j 个人完成的代价，x_{ij} 表示第 i 个任务是否由第 j 个人完成。

利用匈牙利算法可以准确得到最优解，其算法流程如 Algorithm 1所示。

Algorithm 1 匈牙利算法伪代码

输入: 检测框列表 D, 轨迹列表 T, IoU 阈值 θ

输出: 匹配列表 M

1. 构建代价矩阵:

$cost\_matrix \leftarrow$ 计算检测框与轨迹之间的代价矩阵

2. 矩阵预处理:

for 每一行 i **do**

$cost\_matrix[i,:] \leftarrow cost\_matrix[i,:] - \min(cost\_matrix[i,:])$

end for

for 每一列 j **do**

$cost\_matrix[:,j] \leftarrow cost\_matrix[:,j] - \min(cost\_matrix[:,j])$

end for

3. 寻找零元素并标记:

$marked\_zeros \leftarrow$ 使用最少数量的线覆盖所有零元素

4. 调整矩阵:

while 覆盖线的数量不等于矩阵的行数或列数 **do**

调整矩阵，重新寻找零元素并标记

找到步骤 3 中未被一行覆盖的最小元素（称为 k）。从所有未覆盖的元素中减去 k，然后将 k 添加到覆盖两次的元素中。

end while

5. 提取匹配:

$M \leftarrow$ 从标记的零元素中提取匹配对

返回结果: return M

但是在实际追踪任务中，不一定出现正好 n×n 的情况，因此需要进行微调。例如出现 n×m 的情况，解决方法是扩充成 n×n，代价矩阵直接补 0。在具体实现中，SORT 还考虑了其它内容，例如去掉未达到阈值的目标，因此其代码还是进行了许多微调，具体如 Algorithm 2所示。

Algorithm 2 SORT 的数据关联伪代码

初始化:

if 追踪器列表为空 **then**

返回空匹配列表，所有检测框作为未匹配检测框

end if

IoU 矩阵计算:

计算检测框和追踪器之间的 IoU 矩阵

匹配查找:

根据 IoU 阈值将 IoU 矩阵转换为二值矩阵

if 存在一一对应关系 **then**

直接使用索引作为匹配

else

使用线性分配算法找到最佳匹配

end if

未匹配识别:

通过检查哪些索引不在匹配列表中，识别未匹配的检测框和追踪器

过滤匹配:

过滤掉 IoU 低的匹配，并更新未匹配检测框和追踪器列表

返回结果:

返回匹配列表、未匹配检测框列表和未匹配追踪器列表

其中线性分配算法的实现则采用的是 lap 库的 `lap.lapjv` 函数，具体输入输出和使用例如下所示。

cost_matrix 二维数组输入，代价矩阵
extend_cost 布尔值输入，如果为 `True`，会适当地扩展矩阵
const 数值输出，表示总的最小化代价
x 一个数组输出，表示每个行被分配给哪个列
y 一个数组输出，表示每个列被分配给哪个行

```
import lap
import numpy as np
```

```
3
4  def linear_assignment(cost_matrix, extend_cost=True):
5  if extend_cost:
6  # 这里可以根据需要扩展代价矩阵
7  # 例如，如果cost_matrix不是方阵，可以添加无穷大的填充值
8  # 但lap.lapjv通常可以直接处理非方阵，所以这里简化处理
9  pass
10 row_ind, col_ind, cost = lap.lapjv(cost_matrix)
11 return cost, row_ind, col_ind
12
13 # 示例代价矩阵
14 cost_matrix = np.array([[4, 1, 3], [2, 0, 5], [3, 2, 2]])
15
16 # 调用函数
17 cost, row_ind, col_ind = linear_assignment(cost_matrix)
18
19 print("最小代价:", cost)
20 print("行分配:", row_ind)
21 print("列分配:", col_ind)
```

代码 1-1 lapjv example

1.4 学习总结

分配问题的思考:

1. 分配问题如何解决增减的情况

2. 低置信度如何处理

提出的算法

现有 n 个检查结果，m 个轨迹。

设置低阈值。先检查轨迹，若某一轨迹与所有的检测结果置信度都低于阈值，剔除该轨迹，不再计算，可能被遮挡或者离开感知区域。再检查检测，若某一检测结果与所有的轨迹置信度都低于阈值，剔除该检测，不再计算，可能是新进目标。此时问题更加贴近于一个线性分配问题，利用 Lapjv 算法求解。

第 2 章 测量指标

1. 匹配单个结果

正负例 (positive/negative) 的是依据预测值，真假 (True/False) 是依据实际值。

表 2-1 基本概念

	预测正例	预测负例
实际正例	TP	FN
实际负例	FP	TN

2. 召回率

召回率（Recall）是指在所有实际正例中，被模型正确预测为正例的比例。其数学表达式如下：

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

参考文献

[1] Xin S, Zhang Z, Wang M, et al. Multi-modal 3D Human Tracking for Robots in Complex Environment with Siamese Point-Video Transformer[C]//2024 IEEE International Conference on Robotics and Automation (ICRA). 2024: 337-344.

[2] Wu H, Li Y, Xu W, et al. Moving event detection from LiDAR point streams[J]. nature communications, 2024, 15(1): 345.

[3] Nagy M, Khonji M, Dias J, et al. DFR-FastMOT: Detection Failure Resistant Tracker for Fast Multi-Object Tracking Based on Sensor Fusion[C]//2023 IEEE International Conference on Robotics and Automation (ICRA). 2023: 827-833.

[4] Zhang Y, Sun P, Jiang Y, et al. Bytetrack: Multi-object tracking by associating every detection box [C]//European conference on computer vision. 2022: 1-21.