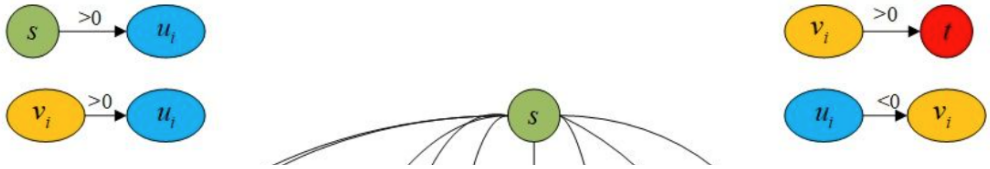
参数分析

加入过渡节点

一个检测目标对应两个节点：入节点 u_i 和出节点 v_i，u_i 和 v_i 之间的弧流量 f_i，代价为 c_i。

为什么一个检测目标对应两个节点：每条边的流量是一，每个节点的流入流量是指向这个节点所有边数的总和，每个节点的流出流量是这个节点指向其他节点边数量的总和，如果不加出节点（过渡节点），每个节点与上一帧所有的节点、下一帧所有的节点、源点 s 、汇点 v 相连，容易造成轨迹的重叠——不同初始节点的路径汇聚到此节点造成路径的重复。在加入过渡节点后，每个节点的流出流量就变成了 1，该节点只能被选择一次，有效的避免了轨迹重叠。



u_i 和 v_i 之间的代价 c_i ： 选择该节点作为轨迹作为轨迹一部分的代价是该节点可能是检测噪声或者错误的检测，用代价 θ 表示该节点丢失或者检测失误的可能性。

连接 v_i 和 下一帧 u_i 的代价 c_i ： 检测该帧被检测目标和下一帧目标的相似程度，计算两帧目标的颜色直方图，它们的分布符合正态分布：

$$P(a_j|a_i) = \frac{N(A_{ij}; A_s, \sigma_s^2)}{N(A_{ij}; A_s, \sigma_s^2) + N(A_{ij}; A_d, \sigma_d^2)}$$

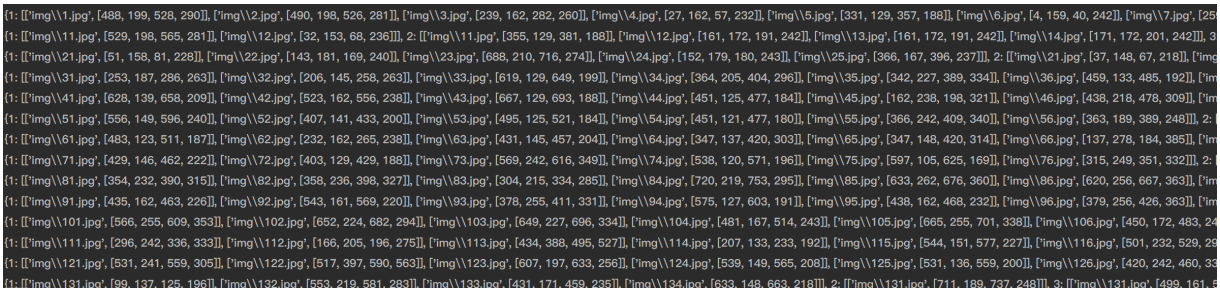
起点 s 到每个入节点 u_i 的代价与汇点 t 到每个出节点 v_i 的代价：这两个参数相同，通过设定初始流量，也就是跟踪轨迹数量，这个值我们都知道是最少是1，最多是总id数，这个问题是一维凸优化问题，采用二分搜索或者斐波那契搜索来进行，可以选择出合适的初始流量。

```
Fibonacci Searching
range = max_flow – min_flow
k = argmin(Fibonacci(k) > range)
s = min_flow
cost = {}
while k > 1:
    u = s + Fibonacci(k - 1)
    v = s + Fibonacci(k - 2)
    if u not in cost:
        cost[u] = MinCostFlow(u)
    if v not in cost:
        cost[v] = MinCostFlow(v)
    if cost[u] == cost[v]:
        s = v
        k = k – 1
    else if cost[v] < cost[u]:
        k = k – 1
    else:
        s = u
        k = k – 2
end
```

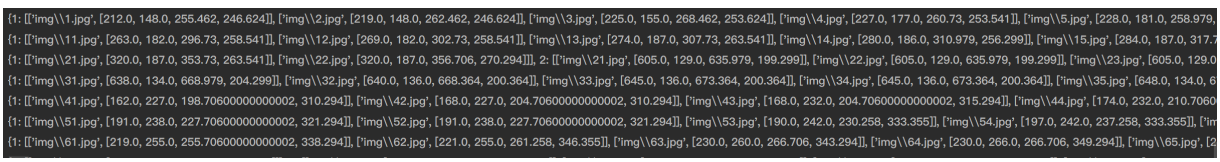
$P_{enter} = P_{exit} = \frac{number\ of\ trajectories}{number\ of\ hypotheses}$ 最终得到该弧的代价。

轨迹选择与轨迹连接

合并两类轨迹



图一.使用最小代价流生成的轨迹



图二.使用重叠法生成的轨迹

将两类轨迹组成集合，从两类轨迹中挑选合适的轨迹：

$$W_{i,j} = W_{\text{appearance}}(t_i, t_j) + \lambda W_{\text{motion}}(t_i, t_j)$$
$$W_{\text{appearance}}(t_i, t_j) = K(\phi(t_i), \phi(t_j))$$

公式一.计算轨迹相似度

轨迹相似度两种指标：$W_{appearance}$ 与 W_{motion} 其中参数 λ 可以调整两种指标所占比例

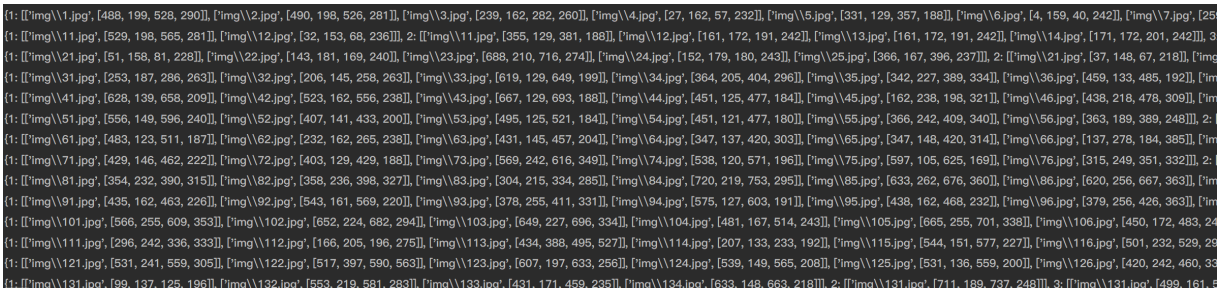
```python
def calc_HS_histogram(image, roi):

    cropped = image[roi[1]:roi[3], roi[0]:roi[2], :]
    #是颜色空间转换函数，p1是需要转换的图片，p2是转换成何种格式
    #cv2.COLOR_BGR2HSV BGR格式转换为色彩空间
    hsv = cv2.cvtColor(cropped, cv2.COLOR_BGR2HSV)
    #计算hsv的色彩直方图
    hist = cv2.calcHist([hsv], [0, 1], None, [180, 256], [0, 180, 0, 256])
    #将结果进行归一化处理
    cv2.normalize(hist, hist, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX).flatten()
    return hist

#对颜色直方图进行巴氏距离比较
def calc_bhattacharyya_distance(hist1, hist2):
    return cv2.compareHist(hist1, hist2, cv2.HISTCMP_BHATTACHARYYA)
```

图三.计算W_appearence的代码

$$d = \sum_{j=0}^{m/2}\sum_{i=0}^{l/2}\left\{y_{s_2+j} - (\dot{y}_{e_1-i})(s_2 - e_1 + j - i)y_{e_1-i}\right\}$$
$$+ \sum_{j=0}^{m/2}\sum_{i=0}^{l/2}\left\{y_{e_1-i} - (\dot{y}_{s_2+j})(e_1 - s_2 + i - j)y_{s_2+j}\right\}$$
$$W_{\text{motion}}(t_i, t_j) = \frac{1}{2}\exp(-d/\partial) \qquad (6)$$

公式二.计算轨迹移动信息相似度

从min-cost flow方法中生成的次级轨迹中的每条轨迹，依次计算其与用重叠方法中生成的轨迹的相似度，选择轨迹相似程度最大的轨迹，将其补充到该条轨迹中（将两条轨迹合并），以此实现轨迹的优化合并。



图四.合并的轨迹

轨迹连接

计算轨迹的颜色相似程度，利用贪心算法取得最大的值，可以实现将次级轨迹连接

但是代码存在一些问题还在调整

轨迹选择

根据两类轨迹的运动相似度 W_{motion} 和外貌相似度 $W_{appearence}$ 来进行轨迹的选择与合并，但是原文中给出的 W_{motion} 公式不好进行计算

$$d = \sum_{j=0}^{m/2}\sum_{i=0}^{l/2}\left\{y_{s_2+j} - \left(\dot{y}_{e_1-i}\right)(s_2 - e_1 + j - i)y_{e_1-i}\right\} + \sum_{j=0}^{m/2}\sum_{i=0}^{l/2}\left\{y_{e_1-i} - \left(\dot{y}_{s_2+j}\right)(e_1 - s_2 + i - j)y_{s_2+j}\right\}$$

$$W_{\text{motion}}\left(t_i, t_j\right) = \frac{1}{2}\exp(-d/\partial) \tag{6}$$

公式一.计算轨迹相似度

根据其引用的参考文献找到了能够进行计算的轨迹相似度公式

$$d = \sum_{j=0}^{\lfloor m/2\rfloor}\sum_{i=0}^{\lfloor l/2\rfloor}\overbrace{\left[\mathbf{y}_{s_2+j} - (\dot{\mathbf{y}}_{e_1-i})(s_2 - e_1 + j - i)\mathbf{y}_{e_1-i}\right]}^{forward\ deviation\ error} + \sum_{j=0}^{\lfloor m/2\rfloor}\sum_{i=0}^{\lfloor l/2\rfloor}\overbrace{\left[\mathbf{y}_{e_1-i} - (\dot{\mathbf{y}}_{s_2+j})(e_1 - s_2 + i - j)\mathbf{y}_{s_2+j}\right]}^{backward\ deviation\ error}.$$

公式二.GMMCP tracker: Globally optimal Generalized Maximum Multi Clique problem for multiple object tracking中的公式

$$\gamma_{motion}(\boldsymbol{V}_s) = \sum_{i=1}^{s}\sum_{j=1}^{s-1}\overbrace{|\boldsymbol{X}_s(i) - \underbrace{[\boldsymbol{X}_s(j) + \dot{\boldsymbol{X}}_s(j).(i-j)]}_{prediction}|}^{deviation},$$

公式三.GMCP-Tracker: Global Multi-object Tracking Using Generalized Minimum Clique Graphs中的公式

将原始公式一修改如下：

$$d = \sum_{j=0}^{\lfloor m/2 \rfloor} \sum_{i=0}^{\lfloor l/2 \rfloor} \overbrace{\left[\mathbf{y}_{s_2+j} - (\dot{\mathbf{y}}_{e_1-i})(s_2 - e_1 + j - i)\mathbf{y}_{e_1-i}\right]}^{forward\ deviation\ error} + \sum_{j=0}^{\lfloor m/2 \rfloor} \sum_{i=0}^{\lfloor l/2 \rfloor} \overbrace{\left[\mathbf{y}_{e_1-i} - (\dot{\mathbf{y}}_{s_2+j})(e_1 - s_2 + i - j)\mathbf{y}_{s_2+j}\right]}^{backward\ deviation\ error}.$$

公式四.修改后的公式

```
#计算两条轨迹的距离相似度
def motion_similarity(track1, track2):
    d = 0
    e1 = len(track1) - 1
    e2 = len(track2) - 1
    #forward deviation error
    for j in range(len(track2) // 2 + 1):
        for i in range(len(track1) // 2 + 1):
            y1 = positions(track2[j][1])
            y2Part2 = positions(track1[e1 - i][1])
            y2Part1 = positions(track1[e1 - i -1][1])
            y2 = [y2Part2[0] - y2Part1[0], y2Part2[1] - y2Part1[1]]
            y3 = positions(track1[e1 - i][1])
            parameter1 = -e1 + j - i
            x = y1[0] - (y2[0] * parameter1 + y3[0])
            y = y1[1] - (y2[1] * parameter1 + y3[1])
            d += math.sqrt(x*x + y*y)
    #backward deviation error
    for j in range(len(track2) // 2 + 1):
        for i in range(len(track1) // 2 + 1):
            y1 = positions(track1[e1 - i][1])
            y2Part2 = positions(track2[j + 1][1])
            y2Part1 = positions(track2[j][1])
            y2 = [y2Part2[0] - y2Part1[0], y2Part2[1] - y2Part1[1]]
            y3 = positions(track2[j][1])
            parameter1 = e1 + i - j
            x = y1[0] - (y2[0] * parameter1 + y3[0])
            y = y1[1] - (y2[1] * parameter1 + y3[1])
            d += math.sqrt(x*x + y*y)
    theta = 1
    return 0.5 * math.exp(-d / theta)
```

经过 W_{motion} 和 $W_{appearence}$ 的比较后将两类轨迹合并成一个轨迹集。

次模最大化

次模函数：一个大集合和一个小集合同时添加一个新的元素 i 时，小集合的增量更大

把轨迹集合分成两类：初始集和候选集，每次从初始集中选一个目标，在下一帧中选其最匹配的（$W_{ij} = 0.7 * W_{motion} + 0.3 * W_{appearence}$最大的）次级轨迹，将两个次级轨迹结合在一起，如果下一段轨迹中没有匹配的次级轨迹

（$W_{ij}max < \lambda$）则认为该轨迹结束，把整条轨迹从轨迹集合中取出，经过实验暂定 $\lambda = 0.65$

如果初始轨迹集为空了，则选下一个不为空的轨迹片段集作为初始集，直到最后一段

经过轨迹选择得到了长的轨迹集合

遮挡处理

次模最大化后的轨迹集中有些轨迹因为遮挡或者检测效果不好造成了长轨迹分成多段小轨迹，遮挡处理就是把这些小轨迹连接在一起。

小轨迹的连接有两个因素的限制：

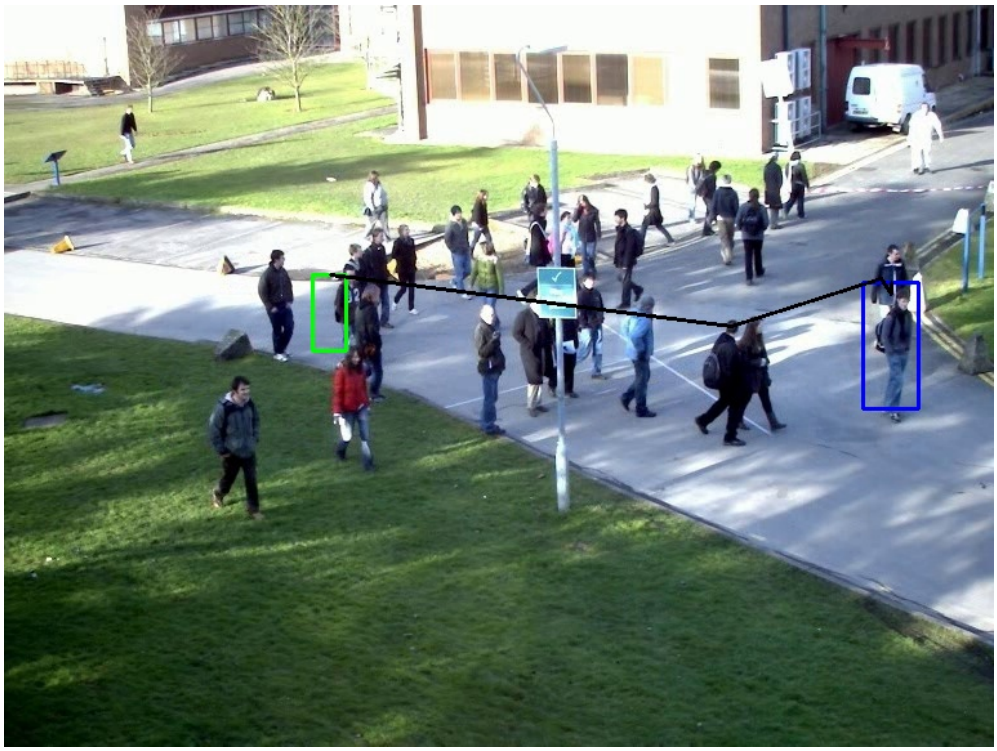
1. 两条轨迹的 $W = 0.7 * W_{motion} + 0.3 * W_{appearence}$ 相似度要大于0.7
2. 两条轨迹之间的断连小于 μ 帧，暂定 $\mu = 20$

再经过一轮轨迹的选择和连接，最终生成完整轨迹集合

```
#连接轨迹
def connect_track(pointEnd, pointStart):
    endNum = int(re.findall(r"\d+",pointEnd[0])[0])
    startNum = int(re.findall(r"\d+",pointStart[0])[0])
    addNum = startNum - endNum - 1
    endX1 = int(pointEnd[1][0])
    endY1 = int(pointEnd[1][1])
    endX2 = int(pointEnd[1][2])
    endY2 = int(pointEnd[1][3])
    startX1 = int(pointStart[1][0])
    startY1 = int(pointStart[1][1])
    startX2 = int(pointStart[1][2])
    startY2 = int(pointStart[1][3])
    addX = (startX1 - endX1) // (addNum + 1)
    addY = (startY1 - endY1) // (addNum + 1)
    addtrack = []

    for i in range(addNum):
        point = []
        imgName = 'img\\' + str(endNum + 1) + '.jpg'
        point.append(imgName)
        endNum += 1
        endX1 += addX
        endY1 += addY
        endX2 += addX
        endY2 += addY
        position = [endX1, endY1, endX2, endY2]
        point.append(position)
        addtrack.append(point)

    return addtrack
```

轨迹展示（暂时的）

将集合中的轨迹连接到一起，绿色框表示开始，蓝色框表示结束，黑色代表轨迹，把轨迹画在该轨迹出现的最后一帧上



图五.其中一条效果比较好的最终轨迹

但是因为第二类次级轨迹（最小代价流）出现一些问题，大部分轨迹效果不是很好，需要进一步进行优化调参

$$
\begin{aligned}
T &= \underset{\mathcal{T}}{argmin} \sum_{\mathcal{T}_k \in \mathcal{T}} -\log P(\mathcal{T}_k) + \sum_i -\log P(x_i|\mathcal{T}) \\
&= \underset{\mathcal{T}}{argmin} \sum_{\mathcal{T}_k \in \mathcal{T}} (C_{en,k_0} f_{en,k_0} + \sum_j C_{k_j,k_{j+1}} f_{k_j,k_{j+1}} + C_{ex,k_{l_k}} f_{ex,k_{l_k}}) + \sum_i (-\log(1-\beta_i)(1-f_i)) \\
&= \underset{\mathcal{T}}{argmin} \sum_i C_{en,i} f_{en,i} + \sum_{i,j} C_{i,j} f_{i,j} + \sum_i C_{ex,i} f_{ex,i} + \sum_i C_i f_i
\end{aligned}
$$

$$
f_{en,i} = \begin{cases} 1 & \exists \mathcal{T}_k \in \mathcal{T}, \mathcal{T}_k \ starts \ from \ x_i \\ 0 & otherwise \end{cases}
$$

$$
f_{ex,i} = \begin{cases} 1 & \exists \mathcal{T}_k \in \mathcal{T}, \mathcal{T}_k \ ends \ at \ x_i \\ 0 & otherwise \end{cases}
$$

$$
f_{i,j} = \begin{cases} 1 & \exists \mathcal{T}_k \in \mathcal{T}, x_j \ is \ right \ after \ x_i \ in \ \mathcal{T}_k \\ 0 & otherwise \end{cases}
$$

$$
f_i = \begin{cases} 1 & \exists \mathcal{T}_k \in \mathcal{T}, x_i \in \mathcal{T}_k \\ 0 & otherwise \end{cases}
$$

轨迹平滑

根据matlab代码：

```
function smoothTraj = smooth_trajectory(centers)

    if (size(centers, 1) > 50)
        wts = [1/36; repmat(1/18, 17, 1); 1/36];
    elseif (size(centers, 1) > 30)
        wts = [1/24; repmat(1/12, 11, 1); 1/24];
    elseif (size(centers, 1) > 24)
        wts = [1/12; repmat(1/6, 5, 1); 1/12];
    else
        wts = [1/6; repmat(1/3, 2, 1); 1/6];
```

```
    end

    % wts = [1/24;repmat(1/12,11,1);1/24];
    if (size(centers, 1) > 8)
        smoothTrajTmp = conv(centers(:, 1), wts, 'valid');
        smoothTrajTmp(:, 2) = conv(centers(:, 2), wts, 'valid');
        diffLength = size(centers, 1) - size(smoothTrajTmp, 1);
        smoothTraj(:, 1) = interp1(diffLength / 2 + 1:diffLength / 2 + size(smoothTrajTmp,
1), smoothTrajTmp(:, 1), 1:size(centers, 1), 'linear', 'extrap')';
        smoothTraj(:, 2) = interp1(diffLength / 2 + 1:diffLength / 2 + size(smoothTrajTmp,
1), smoothTrajTmp(:, 2), 1:size(centers, 1), 'linear', 'extrap')';
        smoothTraj = round(smoothTraj);
    else
        smoothTraj = centers;
    end
```

改写成python代码，实现对轨迹的平滑，且便于计算轨迹 W_{motion}

经过查找文献和代码，确定了计算 W_{motion} 的公式为：

$$d = \sum_{j=0}^{m/2}\sum_{i=0}^{l/2} \mid y_{s_2+j} - [(s_2 - e_1 + j - i)(\dot{y_{e_1-i}}) + y_{e_1-i}] \mid \ + \ \sum_{j=0}^{m/2}\sum_{i=0}^{l/2} \mid y_{e_1-i} - [(e_1 - s_2 + i - j)(\dot{y_{s2+j}}) + y_{s_2+j}] \mid$$

$$W_{motion} = 1/2 * exp(-d/\sigma)$$

遮挡处理

对最终生成的轨迹依次比较，将大于相似度阈值的两条轨迹连接成新的轨迹，再依次对比后续的轨迹，知道把能连接的轨迹全部连接起来。

最终生成轨迹257条，可能是对比轨迹相似度的方法不够成熟，且数据集中大部分目标太过拥挤，导致最终结果不是十分理想，生成轨迹在附件中

$$\begin{cases} Sim(T^i_{trajectory}, T^j_{trajectory}) > \tau_s \\ Frame_i \bigcap Frame_j = \emptyset \\ \min Frame_j - \max Frame_i \leq \tau_f, \ \ if \ \ i \leq j \\ \min Frame_i - \max Frame_j \leq \tau_f, \ \ if \ \ j \leq i \end{cases}$$