# 一、数据存储方式

1. 数据点、带时间的数据点：使用 struct Posi 、 Posi_t。前者用于 road 里的数据点，后者用于 track 里的数据点。
2. 图：使用邻接表，用 vector 数组实现。定义 struct road 存储路段 id，数据点个数，路段长度，开始的点，结束的点以及道路等级。里面还有一个 Posi 型 vector 存储点信息。
3. 轨迹 Posi_t 的 vec 组成的 vec。
4. 格子：长度、数量均可变，用二维 vector 实现
5. 当前的概率与最对应的前置状态：用 map 实现
6. 最短距离：用 map<int,double>型的 vector，每个 map 代表始发点，其中第一个 int 是到达点的 id，第二个 double 是最短路的长度。

# 二、函数介绍

1. cqlt_pp_eu：输入两个点，返回二者的欧拉距离。定义为内联函数。
2. read()：先读取道路信息，忽略道路等级的 string，顺便得到道路的长度、道路最西、最南的值。由于输入数据把两条相反的路放在相邻位置，如果是二者终点、起点相同，则二者为友路。然后读取轨迹信息。
3. initial()：根据 read 得到的经纬度最小值，结合定义的格子大小，计算每条路上每个点应该属于哪个格子，然后把路的 id 插入格子里，同时插入周围的 8 个格子中。插入完成后对格子中去除重复边。
4. no_dunjiao：输入三个点，第一个点是轨迹点，另外两个个是路段起始点。返回一个布尔值，判断路段起始点是否是钝角。都不是则返回 false。
5. cqlt_s：输入三个点，返回这三个点对应三角形的面积。
6. cqlt_posi_to_edge，输入一个点和一个边，利用刚刚的面积和路段距离计算该点到此边的距离，并判断垂线是否在路段上。返回路段和点的距离。
7. cqlt_start_to_posi：输入一个点和一条边，返回这个点到路段开始点的距离
8. Dijkstra()：用优先队列优化的版本。对 shortestlens 经行修改。首先循环始发点，对于每个始发点，创建一个<double,int>型小端优先队列 double 是首发点到顶点为 int 值最短路径长度。每次 pop 一个 pair 出来，判断是否经过 pair 对应的顶点会不会更优。当 q 为空时、最短路径长度已经超过约定值、已经有 50 条最短路时就结束，将结果加入 shortestlens 中。最开始是存路径，用一个单独的函数计算距离。但是这样会带来额外的时间开销。所以直接往 map 里存距离就行，避免重复计算，直接预处理。但是这样就会带来一个问题，就是有可能会访问到不存在的元素，所以需要判断 key 是否在 map 里。
9. cqlt_p2p_onroad：输入 2 点和 2 点对应的匹配上的边，利用刚刚的 Dijkstra 算法得到的路径距离，返回二者在地图上的路径长度。
10. guancegailv：输入一点和一边，根据点和边的距离，以及道路等级，由正态分布概率公式得到观测概率：

$$p(z_t|r_i) = \frac{1}{\sqrt{2\pi}\sigma_z} e^{-0.5\left(\frac{\|z_t - x_{t,i}\|_{great\ circle}}{\sigma_z}\right)^2}$$

11. zhuanyigailv：输入两点两边，计算出两点在地图上的距离 dis，然后计算出转移概率。其中 β 是和 t 有关的超参，随着两点时间差越来越大，β 就越来越大：

$$p(d_t) = \frac{1}{\beta} e^{-d_t/\beta}$$

12. match：输入一条轨迹，一个空的 vector 存储前向边，和一个空的 vector 存储最后的概率。根据轨迹得到所在格子，如果是第一个点就只计算观测概率，之后所有的点还要计算状态转移概率。将遍历每个点的所有边，将状态转移概率与所有前向边概率相乘。然后找到乘积最大的一项，得到对应的前向边与概率，乘观测概率，更新，然后开始下一次遍历。这样结束就得到每一点中所有匹配边的前向边以及最终的概率，这两个就是要修改的“返回值”。
13. print：传入前向边矩阵和最终概率矩阵，找到最大概率的一项，在前向边矩阵中向前遍历，直到最开始。然后输出遍历结果。

# 三、优化

**1.准确度优化：**

1.距离计算：由于 1 度大概对应 111km 实际长度，所以会有个系数，让每个点都乘以这个系数，可以得到相应的优化。(有小作用)

2.反向路：在匹配的时候，由于两条相邻的反向路，有可能会带来观测概率的不准确。于是判断道路方向与前进方向是否相同。如果相反，如果这条路有友路，就采用其友路，如果没有友路，就采用长度第二的路。(作用不大)

3.匹配的边的范围：将原来较大的格子缩小为 1/3，每次匹配不仅将原来的格子放进去，而且把周围 8 个格子的边也加进来（作用较大）

4.输出时检查重复边：由于不会绕圈，所以一条边除了相邻时会重复，距离 2-5 条边如果出现了重复边，那么就应该把这之间的其它边置为当前边。(作用很大)

5.归一化：在匹配过程中，把概率置为 0-1 之间，避免越乘越小，以至于精度不够。(作用不大)

6.道路等级：由于等级越高的道路数量会越少，所以从概率上将，匹配到高等级的道路概率会略小于普通道路。

7.道路限速：根据轨迹的速度大小与道路限速比较，越接近的就转移概率越大。(作用不大，并且会带了较大的内存开销，会爆内存，因此没有启用)

**2.时间优化：**

1.采用 emplace_back 代替 push_back，提高性能。(作用不大)

2.在遍历的时候提前得到 size，而不是每次都调用 size 函数（作用不大）

3.在向格子里插入边的时候直接插在周围 8 个格子里，而不是在匹配过程中加边的时候一个格子一个格子的加边，直接对所在格加边就行（作用很大，不但时间减小，而且内存也显著减小）

4.在向格子里加边完成后对格子里的边去重。(作用非常大，尽管看似

比较蠢，但是由于匹配的边数量急剧减少，避免大量重复计算）

5.格子的大小：如果减小每个格子的大小，就会减少边的数量，就减少时间（作用很大，但是会降低准确率）

6.直接在 Dijkstra 里得到距离，直接预处理出，而不是通过另一个函数计算距离（作用很大）

7.在 Dijkstra 里，结束的判断标志可以适当减小，与格子长度相匹配（作用很大，还会同时降低空间消耗）

8.在对准确率要求不高的时候关闭准确率优化，可以提高速度。

# 四、调节参数

需要调节的参数一共 2 个，分别是 Dijkstra 中的停止距离 l，格子的大小 h,l 和 h 越大，带来的时间开销就越大，但是准确度就越高。
算法可以达到 70、90、93 的准确度。

- 70 准确度下，l=0.0005，h=0.001
- 90 准确度下，l=0.0006，h=0.0016
- 93 准确度下，l=0.0008，h=0.0024

# 五、效率分析

假设：
输入的边数为 e，顶点数为 n，平均每个边有 a 个顶点；
输入的轨迹数为 m，平均每个轨迹有 b 个顶点；
格子数为 x，平均每个格子去重前有 y 个边，去重后又 z 个边。
则：

**时间复杂度：**

1. 读入函数复杂度为 O（ea+mb）
2. 初始化函数复杂度为 O（$ea+x^2ylogy$）
3. 迪杰斯特拉算法复杂度为 O（nzlogz）（由于边数和顶点数大致相等，只会在一个格子大小的范围内 dijkstra）
4. 每次得到观测概率和状态转换概率的复杂度为 O（a），因为要对边逐段分析。
5. 每个轨迹的每个点需要匹配 O（$z^2$）次，每次都要求状态转移概率（除了第一次）以及观测距离，所以每次匹配的复杂度为 O（$z^2ab$）
6. 每次输出的的复杂度为 O（b+z）
7. 一共要匹配 m 次

因此，程序的时间复杂度为:
O（$ea+mb+ea+ x^2ylogy+ nzlogz+mz^2ab+ b+z$）
= O（$ea+m z^2ab + x^2ylogy+ nzlogz$）

**空间复杂度：**

1. 存边的空间复杂度为 O（ea）
2. 存轨迹的空间复杂度为 O（mb）
3. 存最短距离的空间复杂度为 O（nz）
4. 存前向边的空间复杂度为 O（$mbz^2$），概率矩阵空间复杂度为 O（mz）

因此，程序的空间复杂度为：

$O（ea+ mb+ nz+ mbz^2+ mz）$

$= O（ea+mbz^2+nz）$