中国科学技术大学
University of Science and Technology of China

大规模轨迹数据高效聚类算法的实现

吴极端

信息与计算科学 数学科学学院

展示流程

- 问题定义
- 技术路线
- 程序实现
- 我的创新

问题定义：数据模型

- 位置点，由它的经纬度坐标表示
  - P = {latitude, longitude}
- 边，由一串位置点表示
  - e = {P1, P2, P3, … , Pm}

- 轨迹的表示：
  - 轨迹的原始数据：T = {P1, P2, P3, …, Pn}
  - 另一种表示：T = {e1, e2, e3, …, es}

问题定义

- 将所有轨迹数据分成 k 个聚类并从中**各选出**一条代表路径.

- 轨迹间距离度量: EBD(Edge Based Distance) measure

$$EBD(T_1, T_2) = max(|T_1, T_2|) - |T_1 \cap T_2|$$

其中|Tra1 ∩ Tra2|表示两条轨迹公共部分(边)的长度

- 目标：将所有轨迹分成k类，得到使得 k 条图心轨迹到该聚类轨迹的距离和之和最短的 k 条图心轨迹

$$\{\mu_1, \mu_2, ..., \mu_k\} = \underset{\mu_1 \in S_1, \mu_2 \in S_2, ..., \mu_k \in S_k}{arg\,min} \sum_{j=1}^{k} \sum_{T_i \in S_j} Dist(T_i, \mu_j)$$

- 显然，每条轨迹都应该分配到距离最近的图心路径所代表聚类中

技术路线

- 以下实验均在**边长度均为** 1 的前提下进行
- 算法的总体思想沿用了**Lloyds's 算法**，主要分为以下三步
  - Initialization
  - Assignment
  - Refinement

算法 1.1 Baseline Algorithm

Data: 所有轨迹数据, 以及相应生成的轨迹道路网络图数据

Output: k 条图心路径

```
initialization();
Iteration = 1;
while 上次修正过程中存在聚类的图心路经发生了改变或者 Iteration = 1 do
    Assignment() ;
    foreach Cluster_i ∈ Clusters do
        Refinement (Cluster_i) ;
    end
end
```

技术路线

- 主要致力于解决的问题首先是运算的高效性，其次是算法的有效性，即算法迭代收敛结果，也就是目标值的大小

$$\text{Objective Value} = \sum_{j=1}^{k} \sum_{T_i \in S_j} Dist(T_i, \mu_j)$$

- 算法的**复杂度**主要集中在 **Assignment** 和 **Refinement** 步骤
  - Assignment 过程中，对每条轨迹判断离其距离最近的图心路径，将该轨迹分配到对应聚类中
  - Refinement 过程中，**遍历**聚类中**每一条轨迹**，计算以其为图心路经的聚类目标值，选出使得目标值最小的轨迹作为图心路经

技术路线

- Assignment 具体算法

算法 1.2 Assignment

Input: 所有轨迹集合 T 以及所有图心路径集合 P

Output: 所有轨迹分配到的聚类序号

```
1  minDistance = ∞;
2  foreach trajectory_i ∈ T do
3      foreach path_j ∈ P do
4          tempDistance = ComputeDistance (trajectory_i, path_j) ;
5          if tempDistance < minDistance then
6              minDistance = tempDistance ;
7              trajectory_i belongs to Cluster j ;
8          end
9      end
10 end
```

技术路线

- Refinement 具体算法

算法 1.3 以已有轨迹为图心路径的 Refinement

Input: 聚类 S, 由聚类 S 中轨迹数据建立的边直方图 (EH), 长度直方图 (LH), 以轨迹 T 为图心路经的聚类目标值记为 OV(T)

Output: 聚类 S 的图心路经 $path_s$

```
1 建立累计长度直方图 (ALH);
2 minOV = ∞;
3 foreach T ∈ S do
4     OV(T) = ||G_s|| − Σ_{e∈T} EH(e) + ALH(|T|);
5     if OV(T) < minOV then
6         path_s = T;
7     end
8 end
```

技术路线：算法的高效性

- 设总共有 n 条轨迹，k 个聚类，单次距离计算的复杂度为 O(dis)
- Assignment 过程复杂度为 O(nk) × O(dis)
- Refinement 过程复杂度取决于选取的具体算法：
  - 选取聚类中**已有轨迹**作为聚类的centroid path：$O(n^2) \times O(dis)$
  - 其他算法：待分析

- 针对算法的复杂度主要在第二、三步骤，有以下提高效率的切入点
  - A. 提高**单次距离运算效率**：Dist(Tra1, Tra2)
  - B. 提升 Assignment 过程中每条轨迹**属于哪个聚类的判断**效率
  - C. 提升在**已知图心路径**时，计算单个聚类的目标值的效率

技术路线：算法的高效性

- A. 提高单次距离运算效率

$$EBD(T_1, T_2) = max(|T_1|, |T_2|) - |T_1 \cap T_2|$$

  - 利用**有序序列**进行求交集运算
  - 建立利用组成轨迹的边序列 **Inverted Index**，先判断两轨迹**有无交集**，对于交集为空的两条轨迹距离运算可以简化：$EBD(T_1, T_2) = max(|T_1|, |T_2|)$

- B. 提升 Assignment 过程中每条轨迹属于哪个聚类的判断效率
  - 记录每条轨迹到各个图心路径的**距离的上下界**
  - 利用距离测度满足**三角不等式**，事先进行**预判**，**减少**所需**距离计算次数**
  - 例如，若下式成立，则轨迹 Ti 绝对不会从聚类 k 移动到聚类 j 中

$$\text{Lower bound of } Dist(\mu_j, T_i) > \text{Upper bound of } Dist(\mu_k, T_i)$$

技术路线：算法的高效性

- C. 提升在已知图心路径时，计算单个聚类的目标值的效率
  - 对每个聚类**构建直方图**简化目标值计算

$$\begin{aligned}\text{Objective Value in Cluster j} &= \sum_{T \in S_j} max(|T|, |\mu_j|) - \sum_{e \in \mu_j} \|e\| \\ &= \sum_{T \in S_j} |T| + \sum_{T \in S_j'} (|\mu_j| - |T|) - \sum_{e \in \mu_j} \|e\| \\ &= \|G_j\| + ALH[|\mu_j|] - \sum_{e \in \mu_j} EH(e)\end{aligned}$$

  - 其中 ||Gj|| 表示聚类 j 中所有轨迹长度之和，**ALH** 是针对聚类 j 建立的**累计长度直方图** (Accumulated Length Histogram)，针对公式第二行第二个求和项，**EH** 是针对聚类 j 构建的**边直方图** (Edge Histogram)，EH(e) = ||e||，||e|| 表示边 e 在聚类轨迹中出现的次数与边长度的乘积，Sj' 表示聚类 j 中长度小于图心路径的轨迹集合，

技术路线：算法的有效性

- 针对算法有效性的提升提出算法 Centroid Path Extraction Problem (CPEP)

$$\mu_j^G = \arg\min_{\mu_j^G \in G_j} \sum_{T \in S_j} EBD(T, \mu_j^G)$$

  - 其中 μ_j^G 是聚类 G 中一条路径，但**不再限定为已有轨迹**，$|\mu_j^G| \in [l_{min}, l_{max}]$
  - 因为已有轨迹也包括在路径的定义中，所以理论上会使聚类目标值**进一步下降**

- 求此问题是 NP-hard，利用贪心算法得到近似解
  - 以聚类中的边为“种子”，利用**宽度优先搜索**从邻边进行延拓，找到长度在聚类中最小轨迹长度和最大轨迹长度之间的一条图心路径
  - 评估一条路径的“**潜质**”，**优先延拓“潜质”大**的待选路径

技术路线：算法的有效性

- 路径的“潜质”，即未来延拓后的目标值下界，下界越低，路径越有可能被延拓为理想的图心路经
  - 记当前待选路径为 ps, 以 ps 为图心路径，聚类的目标值记为 OV(ps), ps 延拓的“潜质”，记为 LB(ps),

$$OV(ps) = \|G_j\| - \sum_{e \in ps} EH_j(e) + ALH_j[l]$$

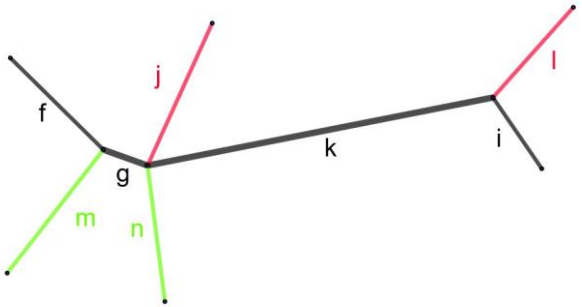
  - 假设 ps' = ps ∪ e_{new}，延拓后

$$\begin{aligned} OV(ps') &= \|G_j\| - \sum_{e \in ps'} EH_j(e) + ALH[l'] \\ &= \|G_j\| - \sum_{e \in ps} EH_j(e) + ALH[l] \\ &\quad - \underbrace{(EH(e_{new}) - ALH[l'] + ALH[l])}_{delta} \end{aligned}$$

  - 取 e_{new} 使得 delta 为可能的最大正数，不断延拓直至 delta 变号
  - 对于有着相同起边和终边的路径，只保留有着更大潜力的待选路径

技术路线：算法合理性

- 前述所有算法均在假定所有边长度均为 1 的前提下进行，而根据论文定义，利用**边的真实长度**显然会使实验**更加合理**
  - 例如下图，黑色轨迹与绿色轨迹，黑色轨迹与红色轨迹同样是公共了一条边，显然黑色轨迹与红色轨迹“相似度”更高，理应距离更小



- 对于Assignment，Refinement (以现有轨迹为图心路径) 的相应改变大同小异
- 对于 Refinement in CPEP 算法，简单的类比并不能得到好的结果

技术路线：算法合理性

- 对于Refinement in CPEP算法，简单的类比并不能得到好的结果，甚至**不能得到收敛的结果**
  - 这里简单的类比是将指仍然试图得到待选路径**未来延拓的目标值下界**
  - 问题变成一个**背包问题**，聚类中的**边是物品**，**边的价值**与边在轨迹中**出现的次数**成**正比**，**物品储量**则是**边的长度**，背包的**剩余空间**则是 $\max_{T \in S_j} |T| - |ps|$

- 变换思路，考虑到我们需要的图心路径是有代表性的，即多次出现的，而受欢迎的路段往往出现在邻近区域，于是引入**加权平均频数** AWF(weighted average frequency)，作为衡量待选路径的指标

$$\mathrm{WAF}(ps) = \frac{\sum_{e \in ps} EH(e)}{|ps|}$$

  - 并为了提高延拓效率，排除了具有相同起边与终边的待选路径中 AWF 最低的那条

程序实现：数据预处理

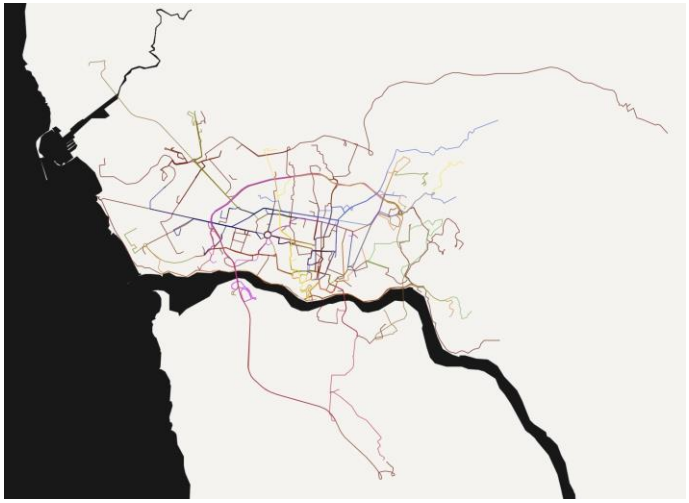
- 原始轨迹数据由一系列 GPS 坐标组成
- 首先进行 **map - matching**，即将一系列 GPS 坐标点与现有的街道、道路对应起来
  - 使用 **GraphHopper API**, 此 API 的算法原理是 **Viterbi 算法**
- 再读取街道地图上的点和边，与轨迹数据结合，建立图
  - 得到最终的轨迹数据由**一系列边序号**组成
  - 边则由一串 GPS 坐标点记录

程序实现：算法主体

- Initialization (事先生成好，用于测试)
  - 随机赋初值
  - k-means++算法生成初值
- Assignment
  - 利用数组记录下界
  - 利用 HashMap 构建Inverted Index
- Refinement
  - 利用 **HashMultiset** 建立直方图
- Refinement in CPEP
  - 利用Priority Queue，在 **compareTo()** 函数中指定特定量作为出队列顺序的标准
  - 利用 HashMap 记录有着**相同起边与终边**的待选路径的出列评判标准

程序实现：可视化

- JavaScript + 百度地图 API



我的创新

- 1. 对论文原公式适用条件的补充.
  - 当轨迹中有较多重复边时，目标值计算会偏不适合运用利用直方图的简化公式进行计算
  - 例如，聚类 S = {path1 , path2}, path1 = {e1, e2, e3}, path2 = {e1, e1, e2}, 并假设所有边长度均为1
    - 无论选取哪条路径作为图心路径，由距离测度的对称性，聚类 S 的目标值都应该等于 EBD(path1, path2) = 3 – 2 = 1
    - 而使用简化后的目标值计算公式时，易得以path1为图心路经的聚类目标值OV1= 0，以及以path2为图心路经的聚类目标值OV2 = 1
  - 由上述例子可知，当轨迹中**重复边比例占到相当部分**时，经过refinement得到的**目标值会偏小**

我的创新

- 2. 利用Assignment过程中每条轨迹的 **“归类” 判断相互**独立和 每个聚类的 Refinement 过程相互独立，将程序并行化

算法 1.4 Baseline Algorithm(并行化)

Data: 所有轨迹数据, 以及相应生成的轨迹道路网络图数据

Output: k 条图心路径

```
1 initialization();
2 Iteration = 1;
3 while 上次修正过程中存在聚类的图心路径发生改变或者 Iteration = 1 do
4     并行化 Assignment() ;
5     foreach 并行化 Cluster_i ∈ Clusters do
6         Refinement (Cluster_i) ;
7     end
8 end
```

我的创新

- 对于Assignment并行化，利用 ConcurrentHashMap 记录移出以及移入的轨迹序号集合以及到新旧图心路径的距离，其中序号集合用 SynchronizedList 实现，在并行化之后进行统一更新

算法 1.5 Assignment(并行化)

Input: 所有轨迹集合 T 以及所有图心路径集合 P

Output: 所有轨迹分配到的聚类序号

```
1  foreach 并行化 Cluster_i ∈ Clusters do
2      foreach 并行化 trajectory_j ∈ Cluster_i do
3          minDistance = ComputeDistance (trajectory_j, path_i);
4          priorDistance = minDistance;
5          foreach path_k ∈ P, k ≠ i do
6              tempDistance = ComputeDistance (trajectory_j, path_k) ;
7              if tempDistance < minDistance then
8                  newClusterId = k;
9                  minDistance = tempDistance ;
10             end
11         end
12         if newClusterId≠i then
13             batchAddTrajectories.put(newClusterId, trajectory_i);
14             batchRemoveTrajectories.put(i,trajectory_i);
15             priorDists.put(trajectory_i, priorDistance);
16             newDists.put(trajectory_i,minDistance);
17         end
18     end
19 end
20 统一进行聚类目标值及轨迹成员, 直方图的更新;
```

我的创新

- 3. 使用边的真实长度进行试验
  - 主要对CPEP问题算法进行改造
  - 不同之处
    - ALH 构建算法更加高效
    - 在Priority Queue中不再使用“潜质” 而是引入加权平均频数

$$\mathrm{WAF}(ps) = \frac{\sum_{e \in ps} EH(e)}{|ps|}$$

    - 使用Buffer来排除不够有优势的待选路径

算法 1.2 My refinement in CPEP

```
Input: 由轨迹数据建立的图 G
Output: min Objective Value (minOV), 图心路经 μ
PQ = ∅, Buffer = ∅, minOV = Double.MAX_VALUE, Iteration = 0;
foreach edge e ∈ EH do
    ComputeAWF(e);
    PQ.push(e, AWF(e));
end
while PQ.IsNotEmpty() and Iteration < MAX_ITERATION do
    (ps, AWF(ps)) = PQ.poll();
    foreach neighbor edge e of ps do
        if EH.contains(e) and e∉ ps then
            ps = ps ∪ e;
            ComputeOV(ps);
            ComputeAWF(ps);
            if OV_ps < minOV then
                minOV = OV(ps), μ = ps;
            end
            construeToken(ps);
            if Buffer.contains(Token(ps)) then
                if AWF(ps) < Buffer.get(Token(ps)) then
                    AWF(ps) = Buffer.get(Token(ps));
                end
            end
            else
                Buffer.add(ps, AWF(ps));
            end
            PQ.add(ps, AWF(ps));
        end
    end
    Iteration = Iteration + 1;
end
```

我的创新：方法对比

- ALH 构建算法的对比

算法 1.3 改进前的 ALH 构建

Input: 聚类 S

Output: 针对聚类 S 的累计长度直方图 (ALH)

```
1 for i = 0 : min_{T∈S} |T| do
2     ALH[i]=0;
3 end
4 for m = min_{T∈S} |T| + 1 : max_{T∈S} |T| do
5     ALH[m] = ALH[m-1];
6     for n = min_{T∈S} |T| : m − 1 do
7         ALH[m]+=LH.count(n);
8     end
9 end
```

算法 1.4 改进后的 ALH 构建

Input: 聚类 S, 其中所有轨迹的长度按照升序记录在 Length Sorted Array (LSA) 中

Output: 针对聚类 S 的累计长度直方图 (ALH)

```
1 for i = 0 : min_{T∈S} |T| do
2     ALH[i]=0;
3 end
4 for i = 1 : LSA.size() do
5     prior = LSA[i-1], present = LSA[i];
6     for m = prior + 1 : present do
7         ALH[m]=ALH[m-1]+i;
8     end
9 end
```

- 改进前 ALH 构建算法的时间复杂度为 $O((\max_{T\in S}|T| - \min_{T\in S}|T|)^2 + \max_{T\in S}|T| + \min_{T\in S}|T|)$
- 而改进后的时间复杂度为 $O(\max_{T\in S}|T|)$

我的创新：方法对比

- 加权平均频数的计算时间也比计算“潜质”更短
  - 计算“潜质”的时间复杂度与轨迹中最长轨迹长度有关

算法 1.5 简单类推得到的 refinement in CPEP 算法

Input: 聚类 S, 待选图心路径 *ps*, 聚类 S 中的边频数降序排列记录在 Frequency Sorted Array(FSA) 里面, 以 *ps* 为 S 的图心路经得到的目标值为 OV(ps)

Output: 图心路经 ps 未来延拓的目标值下界 LB(ps)

```
1  space = max_{T∈S} |T| − |ps|, LB(ps) = OV(ps);
2  for i = 0 : FSA.size() do
3      item = FSA[i], ps' = ps ∪ {item};
4      delta = EH(item) − ALH[|ps'|] + ALH[|ps|];
5      volume = min{|item|, space} ;
6      space = space-volume;
7      if delta>0 or space <=0 then
8          break;
9      end
10     OV = OV + delta, ps = ps';
11 end
```

我的创新：方法对比

- 由下式易知频数的计算时间复杂度则为常数

$$WAF_{new} = \frac{ALH \times |ps| + EH(e_{new})}{|ps'|}$$

  - 其中 $ps' = ps \cup e_{new}$

我的创新：合理化CPEP算法有效性

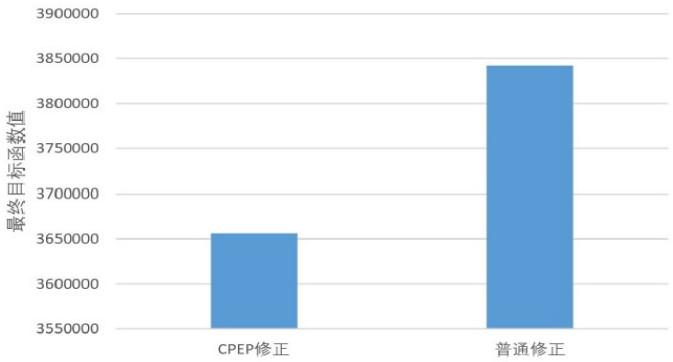


图 5.9　|D| = 1000

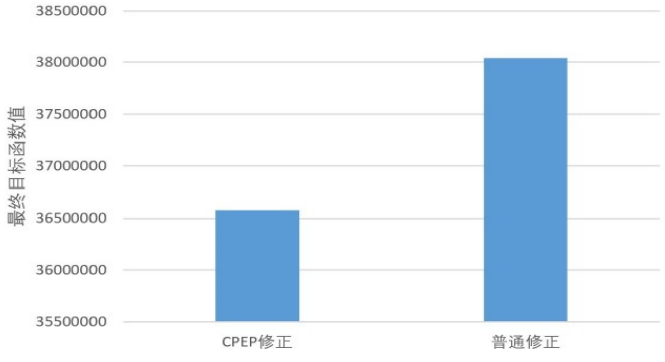


图 5.10　|D| = 10000

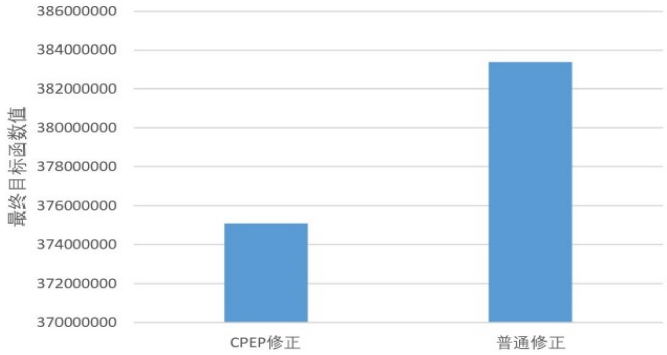


图 5.11　|D| = 100000

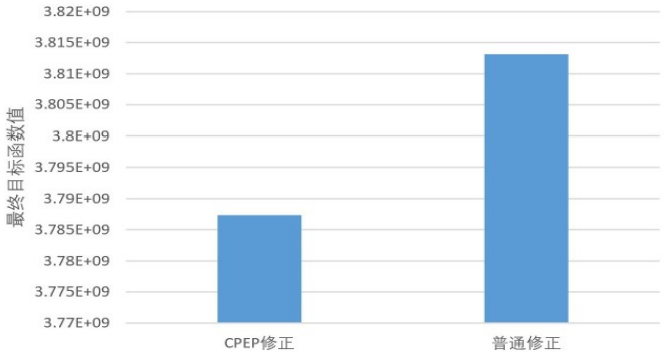


图 5.12　|D| = 1000000

图 5.13　不同数据规模下，两种修正算法得到的最终目标值

我的创新：合理化CPEP算法运行时间

图 5.14　采用普通修正与 CPEP 修正的运行用时对比