申请上海交通大学学士学位论文

相似轨迹查询方法设计与实现

论文作者 戚　文韬

学　　号 5130309593

导　　师 朱燕民教授

专　　业 计算机科学与技术专业

答辩日期 2017 年 5 月 15 日

上海交通大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：_____

日　期：_____年_____月_____日

上海交通大学
学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于

保　密 □，在 _____ 年解密后适用本授权书。

不保密 □。

（请在以上方框内打 ✓）

学位论文作者签名：_____　　指导教师签名：_____

日　期：_____年 _____月 _____日　　日　期：_____年 _____月 _____日

相似轨迹查询方法设计与实现

摘　要

待续

关键词： 上海交大　饮水思源　爱国荣校

A Sample Document for LaTeX-basedd SJTU Thesis Template

ABSTRACT

TBC

KEY WORDS: SJTU, master thesis, XeTeX/LaTeX template

目 录

第一章　绪论

现代社会地理位置获取和移动计算科技进步，促使轨迹数据的大规模发展。这些轨迹数据体现了例如人类，车辆以及动物等移动物体的移动多样性。在过去十几年间，许多旨在处理、管理和挖掘轨迹数据的算法与技术许多应用中有着广泛而重要的应用价值。如今以轨迹数据挖掘为首的轨迹数据处理技术已经日趋系统且规范，从轨迹数据生成，到轨迹数据预处理，再到轨迹数据管理，最后到多样的数据挖掘任务（例如轨迹模式挖掘、轨迹异常检测、轨迹分类等等）。已有轨迹处理和轨迹挖掘的技术在相互应用中有着重要的联系与关联，轨迹数据转化成其他轨迹形式，例如图、矩阵和张量的方法也在越来越多的轨迹数据挖掘和机器学习领域有着常见的应用。

轨迹从概念上定义是一个移动物体的移动轨迹，轨迹数据可以用于许多领域的复杂分析。例如，公共交通系统可以应用过去时刻的轨迹数据分析交通流量模式并找出致使交通拥塞的原因；生物领域的动物长途迁移轨迹或是短途移动变化可以为人类提供宝贵的数据分析人类活动对生态环境的影响程度；还可以通过分析数据预测城乡车辆移动情况并及时提供符合公众出行的公共交通支持。其他应用领域也包括了路径优化设计，公共交通安全管理和基于兴趣点的用户个性化服务。

基于以上应用情景，轨迹数据挖掘在计算机科学、社会学和地理学领域都变得愈发重要。在轨技数据挖掘领域研究从深度和广度都已经取得了不错的成果，从图1–1可以看出当前轨迹数据挖掘与处理的基本研究步骤。本课题相似轨迹查询方法设计与实现主要基于其该范例中的轨迹预处理与轨迹数据索引与获取这两个领域中已存在的方法，并结合自己的理解和数据的格式实现改善和创新。

1.1　相似轨迹查询

大量空间轨迹数据为我们提供了分析移动物体移动方式的可能性，这种移动方式的分析可以体现出单个轨迹所包含的某种特定移动方式或是一组轨迹所共享的相似移动方式。通常情况下相似轨迹查询是基于时空关系的查询，除此之外有些情况下一些相似轨迹查询会增加特定的查询条件，例如最快速度、偏移方向或是在规定的时间段内经过特定地理区域等等条件。在相似轨迹查询中缺少时间维度参数 (时间戳或是时间段) 是可以接受的，加入时间参数的相似轨迹查询本文将他们视为其中的一种特殊情况处理。

1.1.1　轨迹查询概念简介

完成在轨迹数据库中复杂的轨迹查询操作是复杂且费时的操作，因为轨迹数据库的规模一般是非常庞大的。因此，轨迹数据库的一个重要点事支持高效的轨迹索引以加速轨迹插叙过程。通常情况下，时空数据的索引技术是空间数据索引辅以时间度量参数。轨迹查询既关注经过的地理位置的拓扑位置顺序，也关注空间物体之间的距离度量，从简单的欧式距离度量到复杂的轨迹之间相似性。从大体上说，如今的轨迹查询依照时空关系分为三类:1）$P\text{-}query$，查询满足特定轨迹段或者时空关系的兴趣点或者查询针对某些兴趣点满足时空关系的轨迹；2）$R\text{-}query$，根据给定的时空区域查询

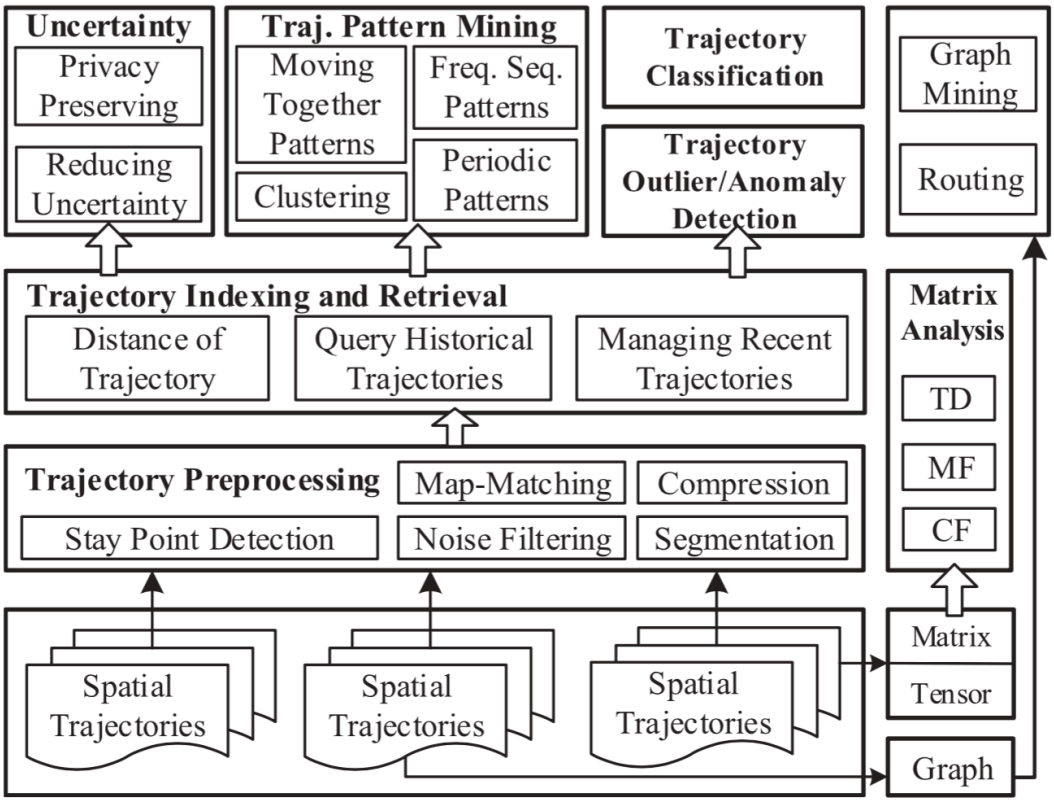

图 1–1 轨技数据挖掘范例

Fig 1–1 Paradigm of trajectory data mining

轨迹或者给定轨迹查询目的区域，3）T-$query$, 查询在一组轨迹数据集中查询相似轨迹或在给定的距离阈值内查询轨迹。

1.1.2 相似轨迹查询应用现状

相似轨迹查询主要是基于上述的轨迹查询方法中 P-$query$ 和 T-$query$ 展开的。

P-$query$ 主要应用在给定地理位置点后找到满足时空关系的轨迹或者轨迹段。单点轨迹查询找到针对某一给定地理位置点的最近轨迹。多点轨迹查询在给定一组地理位置点集后在轨迹数据里中找到能在地理位置意义上连接查询点集的多条轨迹。前者用以找到某一地理位置范围内的潜在轨迹。后者在制定行进轨迹路线中有着很好的应用。T-$query$ 通常通过聚类或分类轨迹在轨迹数据库中查询轨迹。轨迹分类和聚类算法在许多应用中有着广泛的应用，例如基于移动物体特征的轨迹测或是分析路网流量结构，在轨迹集合发现共同的子轨迹以及查询与目标轨迹在欧式距离上最接近的轨迹集合。

基于 P-$query$ 的查询主要是衡量点到轨迹的中最近点的距离。目前也常通过拓展这一思路当多点的 P-$query$ 查询以评价一条轨迹连接多个查询点的好坏。在 T-$query$ 这一查询类型方面则有很多

较为成熟的方法主要的不同在于他们各自的相似距离函数的定义，例如动态时间规划轨迹方法 (Dynamic Time Warping)[ref]、最长公共子序列方法 (Longest Common Subsequence)[ref]、基于编辑代价的方法 (Edit Distance With Real Penalty)[ref] 和基于序列编辑距离的方法 (Edit Distance on Real Sequences) 等等。这些方法在初期主要应用于时序相关的数据上，但是由于轨迹在某种意义上可以看成是多维度上的时序数据，上述的相似距离方法则可以应用上轨迹数据上。

1.1.3 相似轨迹查询方法设计概述

本文研究的相似轨迹查询方法主要基于位置定的查询，即查询主要是基于一组有序或是无序的地理位置点。查询的首相要目标是从轨迹数据库中找出连接查询位置点的 k 条最佳连接轨迹 (K Best-Connected Trajectories) 使得这 k 条轨迹能够在地理位置上连接给定的未指定。不同于传统形状或其他查询标准通过给定一条轨迹的相似轨迹查询，本文的相似轨迹查询主要针对于所查找到的轨迹对于给定的一组轨迹点连接性的优劣。

如图1–2所示，通过点击地图或图像地理解码给定一组地理位置点 (图中点注释)，我们可以从数据库中获取找一条能够连接给点地理位置点原始轨迹 (图中线注释)，该实例体现出本相似轨迹查询方法在能够在包括旅游路线规划等新兴应用中更好地服务用户。与此同时，这种相似轨迹查询能还能在以上场景有所应用：旅行社或自由行游客对出行经典的路径规划；动物园能调查出动物对到某些特定地点的最短路径；交通运输部门对本地市民城乡情况的规划。

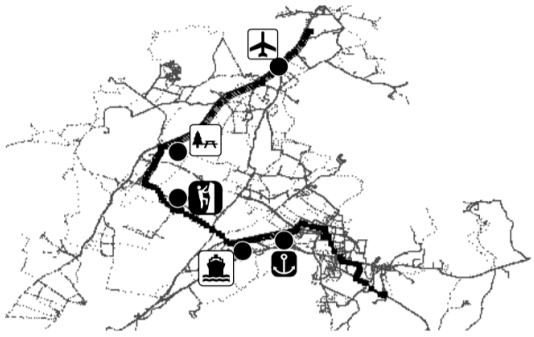

图 1–2 基于位置点的相似轨迹查询
Fig 1–2 Similar trajectory search by locations

大体上,k 条最佳连接轨迹查询基于的地理位置点需要具备必要的经纬度信息 $(latitude, longitude)$。这些经纬度位置地点可以是旅游景点，不明确的沙滩或是任何一处地理坐标。用户可以通过决定轨迹连接有序或是无序以决定查询结果。例如一个简单的查询包括三个地理位置点 A,B,C

$$A_{(37.2601,122.0941)}, B_{(37.2721,122.0648)}, C_{(37.3344,122.1538)}$$

其中 (37.2601, 122.0941) 代表 A 点的经纬度地理坐标。如果查询附带有序条件，则查询轨迹结果应保留轨迹点之间的相对顺序性，即 $A \rightarrow B \rightarrow C$。传统的相似轨迹查询方法显然无法解决查询

结果与查询条件之间有序一致性。另外，本文所提出的轨迹查询方法基于任意的地理位置点使得查询模式更加多样和灵活。从本质上而言，这种这种查询方法的设计基于传统的单点查询方法以寻找针对某一位置的最为临近轨迹。

为了实现这一相似轨迹查询方法，本文首先对一组给定地理位置点中的每一个点进行基于点的查询以从数据库中找到最近的轨迹点。如果查询结果存在，通过对每一个结果的汇总所得出的轨迹从理论上而言是最接近查询点的一条轨迹且对给定的地理位置点具有良好的连接性。从这个思路出发，本文拓展 K 最近邻 (k-NearestNeighbor algorithm) 算法并提出增长性 K 最近邻 ($Incremental\ k\text{-}NN\ algorithm$) 算法。该算法增长性获取每个查询位置点的最近轨迹并不断检查以查询到的轨迹。通过自定义的轨迹相似性上界与下界，借鉴备选和筛选的算法设计思路 (candidate-refinement) 来进行优化与剪枝。利用 R 树 (R-Tree) 作为地理位置点的索引结构，算法实现过程中根据具体计算机情况和性能需求选择性使用最好优先搜索 (best-first) 方法或是深度优先搜索 (depth-first) 方法进行查询。

1.2 论文大致结构

本毕业论文主要结构为：本章介绍轨迹查询大致概念与相似轨迹查询现状与方法大体设计；第二章介绍实现相似轨迹查询方法设计的相关工作；在第三章详细讨论算法实现细节与优化问题处理；第四章中讨论相似轨迹查询和分布式结合具体过程与算法实现；实验过程和结果会在第五章中予以描述并在第六章为本文做结论。

1.3 本章小结

1.1.1、1.1.2 和 1.1.3 三节内容已经初步介绍了相似轨迹查询这一概念和其相关背景，由于目前的在轨迹数据处理已经系统和规范的处理流程，轨迹数据挖掘这一领域的方法技术也已经较为成熟且丰富，通过学习传统的相似轨迹查询方法和他们各自的应用经验，本文所提出的方法在已有成果的基础上进行进一步的创新与优化，便可以使得相似轨迹查询方法与传统的相似轨迹查询有着较大的不同，且具有特定查询环境上的查询优势与性能优化。

本文通过拓展如今最为基本的 k 最近邻数据挖掘技术，以实现基础的相似轨迹查询方法为基本理论目的，移植单机运行代码至分布式环境系统为应用目标，开展毕业设计课题。

第二章　相关工作

2.1　相似度方程定义

相似轨迹查询工作在某种意思上和时序数据相似查询共享一些方法定义。相似轨迹查询的首要步骤通过某种选定轨迹与轨迹点间的距离度量来定义相似度 (或称为距离) 方程，之后是设计高效的查询过程算法来解决从大规模数据库中找到符合要求的备选轨迹。定义相似度方法在过去有许多深度的讨论，之前的工作有通过利用离散傅里叶变化（Discrete Fourier Transform）[Ref bylocation2] 将轨迹数据转化为多维空间上的点，任何通过比较这些点在特征空间上的欧式距离来比较轨迹数据时间的相似性。之后有科研人员在此工作成果的基础上通过改善实现子轨迹的匹配查询，并验证了离散小波变换（Discrete Wavelet Transform）的可行性。切比雪夫多项式（Chebyshev polynomials）在轨迹近似和索引方面有被证明是可应用的。但是这些方法的前提调前是需要轨迹上时序上具有相同的长度，因此这些转变返程所提供的相似度方法不适用与本文所提出的相似轨迹查询。

Definition		
DTW(R,S)	=	$\begin{cases} 0 & \text{if } m = n = 0 \\ \infty & \text{if } m = 0 \text{ or } n = 0 \\ dist(r_1, s_1) + min\{DTW(Rest(R), Rest(S)), & \text{otherwise} \\ DTW(Rest(R), S), DTW(R, Rest(S))\} & \end{cases}$
ERP(R,S)	=	$\begin{cases} \sum_1^n dist(s_i, g),\ \sum_1^m dist(r_i, g) & \text{if } m = 0,\ \text{if } n = 0 \\ min\{ERP(Rest(R), Rest(S)) + dist(r_1, s_1) & \\ ERP(Rest(R), S) + dist(r_1, g), & \text{otherwise} \\ ERP(R, Rest(S)) + dist(s_1, g)\} & \end{cases}$
LCSS(R,S)	=	$\begin{cases} 0 & \text{if } m = 0 \text{ or } n = 0 \\ LCSS(Rest(R), Rest(S)) + 1 & \text{if } \forall d, \lvert r_{d,1} - s_{d,1} \rvert \le \epsilon \\ \max\{LCSS(Rest(R), S), LCSS(R, Rest(S))\} & \text{otherwise} \end{cases}$
EDR(R,S)	=	$\begin{cases} n,\ m & \text{if } m = 0,\ \text{if } n = 0 \\ \min\{EDR(Rest(R), Rest(S)) + \text{subcost}, & \text{otherwise} \\ EDR(Rest(R), S) + 1, EDR(R, Rest(S)) + 1\} & \end{cases}$

图 2–1 相似度函数定义[1]

Fig 2–1 Definition of distance functions

图2–1是典型且常用的相似度方程定义。这些方程根据自身特点与优势应用于不同的场景中，包括欧氏距离方程（Euclidean Distance），动态时间规整（Dynamic Time Warping），最长公共子序列算法（Longest Common Subsequence），基于编辑代价的方法（Edit Distance With Real Penalty）和基于序列编辑的距离方法（Edit Distance on Real Sequences）。动态时间规整（DTW）方法在比较轨迹之间相似性的过程中采用了时间偏移（time-shifting）来使得轨迹中的一些点可以尽可能多地重复出现以实

[1] $dist(ri, si)$ = L1 or L2 norm; $subcost = 0$ if $r1, t{-}s1, t$, else $subcost = 1$

现最好效果的校准。但这种方法在原有轨迹数据点出现误差（或称为噪声点）的时候会影响比较的准确性因为所有的点都需要被匹配。相比如动态时间规整方法（DTW），最长公共子序列（LCSS）选择忽略某些点以避免对他们的重排序过程，从结果上而言这种方法舍弃了偏离采样的误差点以提高了准确性，但需要人为预定距离阈值以判断什么数据属于误差点。基于编辑代价的距离方法（EDR）与 LCSS 方法类似，他们最初提出是为了解决字符串匹配问题，在轨迹数据匹配这一方面他们均采用一个阈值参数来判断两个点是否匹配，但 EDR 考虑了距离之间的衡量代价以决定是否将两个点进行匹配。在此基础上基于序列编辑的距离方法（ERP）结合 EDR 和 DTW 方法选择固定点进行距离计算。

相似度方法通常根据具体的应用进行具体的选取。但上述的相似度方法主要是基于轨迹与轨迹之前相似度的查询，在本文设计的相似轨迹查询方法上的应用度并不理想，本工作的查询条件是基于一组地理坐标点的查询，并且工作更关注与一条轨迹是否能够很好地连接上给定的一组查询点，从而提供基于轨迹点的相似轨迹结果。因此，在这样的情景下，我们需要定义一个新的相似度方程。

2.2 轨迹数据预处理

2.2.1 WGS84 坐标系统转换至 GCJ-02 坐标系统

WGS84（World Geodetic System1984）坐标系统是 GPS 数据所基于的坐标系统，这一坐标系是通过世界卫星观测站所检测到的地理坐标。这一坐标系并不能直接应用在中国国家的地图坐标显示中，因为中国国家会测局在地理信息系统中使用的是基于 GCJ-02 的坐标系统，这一坐标系统也称为火星坐标系统。如果直接将 WGS84 坐标数据应用于使用 GCJ-02 的地图显示接口，则会造成 100 米到 700 米范围内的显示误差。同理，用户使用 GCJ-02 地图点击获得的地理位置查询点也会在相似轨迹查询中因为与 WGS84 坐标系统的偏差因素造成查询结果的不准确性。在轨迹预处理最开始先将 WGS84 坐标数据根据已有的算法[1]参考转换成 GCJ-02 系统下的坐标

2.2.2 轨迹数据简化

轨迹数据预处理中，轨迹数据的简化（或压缩）是比较重要一步。轨迹数据简化主要是指在保证轨迹的可利用性与大致准确的同时减少轨迹的点数目，以达到轨迹数据的传输、处理和存储上减少开销的目的。在本文的应用场景中，我们首先采用 *Douglas–Peucker* 算法来完成我们的轨迹简化任务。该算法的主要思路在于将通过分而治之，将曲线轨迹表示成一系列点的方法，从而减少点的数目。如今 GPS 的数据采样通常较为频繁，因此在我们对轨迹处理的范围上来所，我们可以近似地将我们所运用的数据集中的轨迹看成是一条连续的曲线，通过 *Douglas–Peucker* 算法以及我们人为设定简化阈值，我们可以高效且合理地进行轨迹简化。

算法2–1在首先连接轨迹首尾两点 $Traj[0], Traj[Traj.length]$，遍历轨迹一遍得到离线段距离最大的点 $Traj[index]$，计算该距离并与预先设定的阈值 ϵ 比较。如果大于阈值 ϵ，则将轨迹以

[1] https://github.com/googollee/eviltransform

算法 2–1 Douglas-Peucker 算法

输入: 一条原始轨迹数据 $Traj$, 简化阈值 ϵ

输出: 简化后的轨迹 $Traj'$

```
dis_max ← 0;index ← 0
for i = 1 to Traj.length − 1 do
    temp_dis ← Traj[i]'s perpendicular Distance to Line(Traj[0], Traj[Traj.length])
    if temp_dis > dis_max then
        dis_max ← temp_dis;index ← i
    end if
end for
if dis_max > ϵ then
    half_left ← Douglas-Peucker(Traj[0 : index], ϵ);
    half_right ← Douglas-Peucker(Traj[index : Traj.length], ϵ);
    res ← half_left ⋃ half_right;
else
    res ← {Traj[0], Traj[Traj.length]};
end if
return res;
```

$Traj[index]$ 为中点分为两端，迭代重复上述工作；如果小于阈值 ϵ，则直接将线段段作为曲线的近似以做简化。当曲线完成上述任务，依次连接处理好的子线段，完成轨迹简化任务。

2.3 轨迹数据索引与获取

空间数据结构对从一个大规模轨迹数据集中获取特定轨迹数据是十分重要的。效率问题是查询大规模数据库或数据集来获取信息的首要考虑因素。而查询效率十分依赖于合理的轨迹索引。轨迹数据根据数据特点的不同姓对索引技术也有着特殊的要求。目前主流的索引技术主要有三类：1）基于空间维度的索引，利用 R 树（R-tree）索引进行查询。通过 3DR 树（3D R-tree）或者 STR 树（STR-tree）进行带有时间维度的查询；2）利用多版本的数据结构，根据特定情况使用 MR 树（MR-tree）、HR 树（HR-tree）、MV3R 树（MV3R-tree）等等；3）将空间划分网格结构然后对应每个网格建立对应的空间索引，这类数据结构包括 MTSB 树（MTSB-tree）和 SETI。本文中我们使用 R 树这一最基本的数据结构，其满足我们对算法的实现需求。

R 树数据结构在空间数据库中应用广泛，许多轨迹索引结构大体上是基于 R 树进行拓展。R 树结构是一个平衡树结构，R 树中的每一个节点代表包含其所有子节点一个区域，这个区域通常被称为最小区域箱（Minimum Bounding Box）。节点中的每一个数据体指向对应的子节点的最小区域箱信息。R 树搜索的关键字是最小区域箱中的每一个节点。如图2–2所示的是 R 树数据结构的 2 种表现形式。在2–2(b) 中我们看到树结构而图2–2(a) 描述了数据和最小边界箱是如何分布在空间中的。

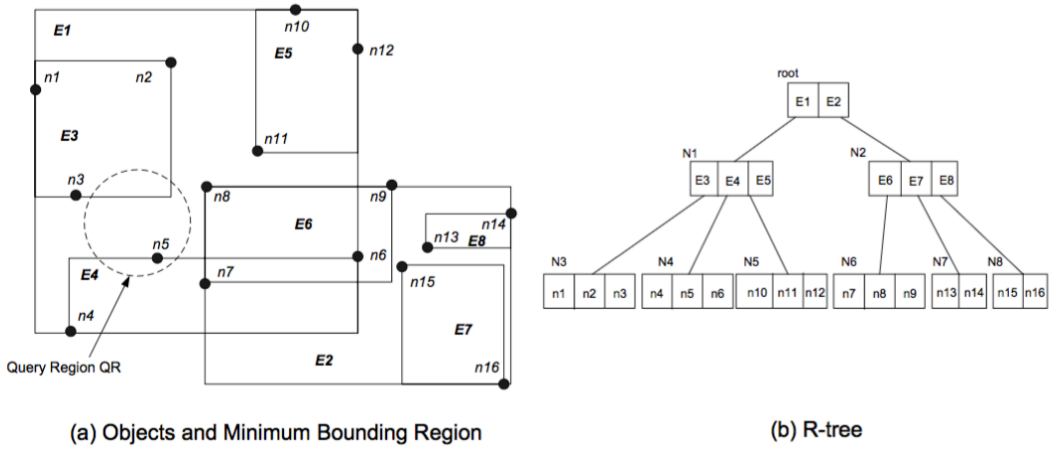

图 2–2 R 树数据结构举例

Fig 2–2 Two views of an R-tree example

在图2–2中，根节点有两个数据体 $E1$，$E2$，分别对应子节点 $N1$，$N2$。$N1$ 代表的最小边界箱包含了其子节点 $N3$、$N4$、$N5$ 以及数据体 $E1$ 所具有的最小边界箱信息。值得注意的是空间点的体现只有在 R 树的叶节点上。R 树可以应用在范围查询和近邻查询中。本文主要使用 R 树近邻查询这一属性。给定一个查询点，R 树可以通过最优优先搜索（best-first）和深度优先搜索（depth-first）两种树遍历策略找到在数据集中最接近查询点的数据。在两种搜索策略中，查询点和每一个最小边界箱的距离被定义为变量 $mindist$。之后的搜索过程基本遵从两种搜索策略各自算法。

在 R 树中插入一个新的节点大致需要以下几个步骤。当有新的轨迹数据需要被添加到已有的 R 树种，首先为待插入的轨迹数据点找到一个合适插入的子节点中。再寻找叶结点的过程中我们会选择符合最小边界范围且对 R 树扩展度最小的一个叶结点。然后若找到 R 树叶结点数据溢出，那么我们需要对叶子结点进行分裂操作；若没有溢出，则可以将待添加的轨迹数据加入到当前已经找到的叶子节点中。最后对 R 树进行变换向上传递并对树高进行增高以完成插入操作。删除操作近似于插入过程的逆过程，在此不予以赘述。

2.4 本章小结

本章节中，本文通过图标和伪代码讨论了相似轨迹查询方法的设计与实现的基本相关工作，对本文所应用的基本定义、处理思路和数据结构有了一个初步的了解。根据本文场景定义个性化的相似度方程后，我们在查询阶段需要通过多点输入的条件下进行归集查询。由于输入数据的轨迹点数目相对较少，我们可以借助已有的数据结构进行空间距离上搜索。利用 R 树和基于 k 最邻近的进行方法拓展是本文实现算法的基本思想，以快速的搜索并获取数据。根据上述本文工作相关工作描述，我们可以得出实现本文算法的先决条件目前都是基于只有已有的成熟工作。在下一章节中，本文将开始对相似轨迹查询方法进行理论讨论。

第三章　相似轨迹查询方法实现

3.1　k 最佳连接

一个轨迹数据库中存储了大量的原始车载 GPS 轨迹或是已经预处理过的车载 GPS 轨迹。这里的轨迹由一系列的地理位置点组成 $\{p_1, p_2, p_3, \cdots, p_m\}$，其中 p_i $1 \leq i \leq m$ 代表一个由经度和维度构成的地理位置点而 m 代表轨迹中点的数目。本文所定义的 k 最佳连接查询（k Best-Connected Rrajectory Query）的输入由一组查询点 Q 组成。q_j $1 \leq j \leq n$ 和 p_i 定义相同，其中 n 是查询点的数目。这里用户可以选择选择是否在查询中指定轨迹连接依照查询点的先后顺序，即是否选择查询有序性。若选择查询有序性，则查询点集 Q 为认为是从 q_1 到 q_m 有序点集。

$$Q = \{q_1, q_2, q_3, \cdots, q_n\}$$

在搜索最好连接轨迹这一上下文中，相似度方程的定义需要和传统方法有所不同，在这里我们将相似度方程定义为一条轨迹连接查询点的好坏程度。因此，本文首要考虑一条轨迹到每一个查询点的距离，我们简要定义距离一个查询点 q_i 到一条轨迹 $R = \{p_1, p_2, p_3, \cdots, p_m\}$ 的距离为 D_q, 即

$$D_q(q_i, R) = \min_{p_j \in R} \{D_e(q_i, p_j)\} \tag{3–1}$$

式3–1中，$D_e(q_i, p_j)$ 是指查询点 q_i 和轨迹点 p_j 之间的欧氏距离，因此通常意义上相似度距离 D_q 代表从查询点 q_i 到轨迹上任一一点距离的最短距离。当我们找到轨迹上一点 p_j 是离查询点 q_i 的最短距离点时，我们将 $< q_i, p_j >$ 作为最短匹配点对。在无序查询点击中，我们定义查询点集 Q 和轨迹 R 之间的相似度为 $Sim(Q, R)$。

定义 3.1.1. 轨迹 $R = p_1, p_2, \cdots, p_n$ 而查询点为 q，$< q, p_i >$ 表示一组匹配点对。对于 $\forall p_i \neq p_j$，$d_e(p_i, q) \leq d_e(p_j, q)$，那么 $< p_i, q >$ 是轨迹 R 到查询点 q 的最短匹配点对。

$$Sim(Q, R) = \sum_{i=1}^{n} e^{-D_q(q_i, R)} \tag{3–2}$$

式3–2将每个查询点对 $Sim(Q, R)$ 的贡献值通过自然对数去反得以体现，即根据自然函数的单调性，查询点离轨迹越近，则 $-D_q(q_i, R)$ 越大，以自然对数为底取幂的值也越大，最后使得 $Sim(Q, R)$ 的值也越大。从用户人为角度和地理语义角度上看，一条轨迹与所有的查询点被定义为相似当且仅当这条轨迹和所有的查询点都十分接近。

图3–1通过距离说明查询点和轨迹之间的匹配关系。如图3–1(a) 所示，查询点 q_1、q_2 和 q_3 分别与轨迹 R 上的轨迹点 p_6、p_4 和 p_7 最近匹配，根据式3–2可以得出，$Sim(Q, R) = e^{-D_q(q_1, p_6)} + e^{-D_q(q_2, p_4)} + e^{-D_q(q_3, p_7)} = e^{-1.5} + e^{-0.1} + e^{-0.1}$。

另一方面，选择查询点和轨迹之前进行有序查询时，顺序性是需要在查询过程中予以考虑。对于查询点 q_i 而言，最近匹配点或许并不在是距离上最近的轨迹点 p_j。因此，相似度方程在此步骤中应该

适当调整。我们再借用图3–1予以说明。假设以下用户场景：用户希望查询出一条以 $q_1 \rightarrow q_2 \rightarrow q_3$ 为顺序的相似轨迹，显然图3–1(a) 中的顺序并不再符合用户需求。实际的有序查询结果顺序如图3–1(b) 是 $p_3 \rightarrow q_4 \rightarrow q_7$。在考虑有序性的相似查询规程中，我们的目标是在保持查询有序性的同时追求每一对匹配点对相似度的最大贡献值，即从图3–1(b) 可以看出 $< q_1, p_3 >, < q_2, p_4 >, < q_3, p_7 >$ 这样的三对匹配点是所有有序匹配对中使得相似度最大的情况。

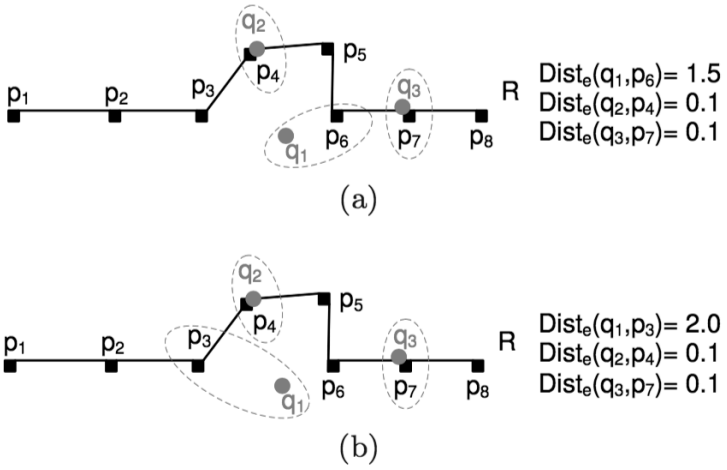

图 3–1 查询点与轨迹之间的匹配

Fig 3–1 Match between query points and trajectory

有序查询的相似性计算和无序查询也有区别。给定一组有序的查询点 $Q_o = \{q_{o1}, q_{o2}, q_{o3}, \cdots, q_{on}\}$ 和一条已有轨迹 R，我们通过递归思想为有序查询重新定义相似度方程为 $Sim_o(Q, R)$，式3–3。其中 $Head(x)$ 函数代表 x 中的第一个点，例如 $Head(Q)$ 是查询点 q_1；同时 $Rest(x)$ 表示 x 去掉 x 第一个点之后剩余的部分，例如 $Rest(Q)$ 代表 $\{q_2, q_3, \cdots, q_n\}$。在式3–3中，通过递归的想法，本文将对 $Sim_o(Q, R)$ 的求最大值问题分为对两个子问题的求解，即分别计算 $Sim_o(Rest(Q), R)$ 和 $Sim_o(Q, Rest(R))$ 的最大值问题。当 $Head(Q)$ 和 $Head(R)$ 的两个轨迹点匹配的时候，我们可以将 $e^{-D_e(Head(Q),Head(R))}$ 提前计算并加入当后面计算的 $Sim_o(Rest(Q), R)$ 之中。在这种情况下，$Head(R)$ 需要为下一轮的比较计算继续保留，因为对于 $Rest(Q)$ 中的查询点来说，$Head(R)$ 依旧有可能成为最佳匹配点。而当当 $Head(Q)$ 和 $Head(R)$ 不匹配的时候时我们则跳过 $Head(R)$ 计算 $Sim_o(Q, Rest(R))$。这种求解思路类似于动态规划的思路，这也为我们再后面优化过程中通过动态规划的来解决这一问题提供了参考。式3–3结合了动态时间规整（DTW）利用重复点和最长公共子序列（LCSS）省略不匹配点的优点来计算相似度方程。

$$Sim_o(Q, R) = max \begin{cases} e^{-D_e(Head(Q),Head(R))} + Sim_o(Rest(Q), R) \\ Sim_o(Q, Rest(R)) \end{cases} \tag{3–3}$$

根据相似度方程，本文可以对 k 最佳连接查询有以下定义 3.1.2.：

定义 3.1.2. 给定一组轨迹集合 $T = R_1, R_2, R_3 \cdots, R_n$、一组查询点 $Q = q_1, q_2, q_3, \cdots, q_n$ 和对应的相似度方程 Sim，k 最佳连接查询则可以从轨迹集合 T 中找到 k 条轨迹 T', 满足式3–4。其中 Sim 根

据用户定义选择是否考虑有序性。:

$$Sim(Q, R_i)_{R_i \in T'} \geq Sim(Q, R_j)_{R_j \in T - T'} \tag{3–4}$$

3.2 相似轨迹查询处理过程

3.2.1 问题概述

轨迹数据为一组有序点集，轨迹 R 可以被表示为 $R = p_1, p_2, \cdots, p_n$，其中 p_i 是轨迹 R 的在时间顺序上的第 i 个轨迹点。对于本文应用而言，查询点集 Q 被定义为一组点集 $Q = q_1, q_2, \cdots, q_m$，且根据具体情况定义是否有序。在根据上文设计的相似性距离和 k 最佳连接定义后，我们将我们相似轨迹查询任务等价转化为 k 最佳连接查询，并产生下面的定义:

定义 3.2.1. 给定已有的轨迹数据集 D，和一条待查询轨迹 R_q。我们通过轨迹简化算法4–5将待查询轨迹 R_q 转换成一组查询点集 Q。通过 k 最佳连接查询方法，从轨迹数据集 D 中获取 k 条轨迹，集合为 $D' = R_1, R_2, \cdots, R_k$，满足式3–5。其中 Sim 根据用户定义选择是否考虑有序性。:

$$Sim(Q, R_i)_{R_i \in D'} \geq Sim(Q, R_j)_{R_j \in D - D'} \tag{3–5}$$

我们称 D' 轨迹数据集合为对于轨迹 R_q 的 k 条最相似轨迹查询结果。

处理上述问题的直观方法可以选择对轨迹集合 D 进行全局搜索，维护一个关键字为相似度大小的优先队列，然后返回结果。但对于系统而言，这样处理数据过于浪费时间。

为解决上述问题，我们首先需要明确我们查询输入的优势在于我们输入的查询点相对于传统的相似轨迹查询方法要小。这是我们能够合理且有效运用空间索引分别对每一个查询点应用近邻查询，并合并查询结果生成 k 最佳相似轨迹的基本前提。这一方法的搜索复杂度相对于查询点集来说相对恒定。因此，当得出一条轨迹对于查询点集的最近轨迹点便为我们计算该条轨迹与查询点集相似度的上下界提供可能。本文中使用 R 树索引并搜索近邻的轨迹点，当我们找到关于某个查询点 q 的最近轨迹点 p，那么包含 p 轨迹点的轨迹一定是离该查询点 q 最近的一条轨迹。

其次在设计搜索框架过程中，我们借鉴备选和筛选（candidate generation and refinement）这一思路。这一思路最初提出为在分布式系统中进行 k 最大数据处理。首先每个子系统的数据按从大到小排列然后并行合并后进行筛选，选出最好的 k 个结果。这一思路结合 k 最近邻查询符合本文对算法设计的要求，我们对查询点集 Q 中的每一个点都进行查询并将他们的结果合并生成暂时的轨迹备选集，然后通过筛选的方法选出最相似的 k 条轨迹。问题的关键在于如何进行轨迹备选集的搜索，以及如何保证 k 条最相似轨迹的完备性。

本文的相似轨迹查询方法利用 R 树数据结构，通过简单的 k 最近邻算法并在其基础上进行有效深度拓展和具体实践优化实现 k 最佳连接查询以实现相似轨迹查询。表3–1提供了本文需要的基本符号及其注释。

符号标记	符号注释
N	一条轨迹的轨迹点总数目
m	一组查询点总数目
$D_e(q_i, p_j)$	点 q_i 和点 p_j 之间的欧氏距离
$D_e(q_i, p_j)$	点 q_i 和点 p_j 之间的欧氏距离
$D_q(q_i, R)$	点 q_i 和轨迹 R 之间的最短距离
C	轨迹备选集
ϵ	TBD
r	λ-NeareatNeighbor 搜索半径
ρ	轨迹点密度
ξ	查询点 q_i 对相似度上界贡献度
μ, υ	优化搜索权值

表 3–1 本文符号列表及其对应注释
Table 3–1 A list of notations and explanations

3.2.2 生成轨迹备选集

将预处理的轨迹点存储在 R 树之中后，我们可以有效地通过 k 最近邻搜索方法来找到离某一查询点 q 最近的一条轨迹。假设现有一组查询点 $Q = \{q_1, q_2, \cdots, q_m\}$，进行无序的相似轨迹查询。我们首先对这一组查询点中的每一个查询点进行 λNN 查询并得到结果如下:

$$\begin{aligned}
\lambda NN(q_1) &= \{p_1^1, p_1^2, p_1^3, \cdots, p_1^\lambda\} \\
\lambda NN(q_2) &= \{p_2^1, p_2^2, p_2^3, \cdots, p_2^\lambda\} \\
&\cdots \\
\lambda NN(q_m) &= \{p_m^1, p_m^2, p_m^3, \cdots, p_m^\lambda\}
\end{aligned}$$

通过每个查询点根据 λNN 算法生成的结果，我们根据结果中的点生成我们的轨迹备选集。包含 $\lambda NN(q_i)$ 中至少一个轨迹点的轨迹被加入到轨迹备选子集 C_i 中用于之后生成 k 最佳连接结果。在这一步我们需要保证轨迹备选子集 C_i 的基数应该小于等于 λ, 因为多个属于 $\lambda NN(q_i)$ 的轨迹点有可能属于同一条轨迹。之后我们合并所有由 $\lambda NN(q_i)$ 查询结果得出的轨迹备选子集以获取一个包含 f 条轨迹的轨迹备选集 C

$$C = \bigcup_{i=1}^{m} C_i = \{R_1, R_2, \cdots, R_f\}$$

对于轨迹备选集 C 中的每一条轨迹 $R_x(1 \le x \le f)$ 而言，R_x 必须包含至少一个离对应查询点距离在固定范围内的轨迹点。即假使 R_x 属于某一轨迹备选子集 C_i(C_i 为轨迹备选集 C 的子集)，那么 $\lambda NN(q_i)$ 中的应该包含轨迹 R_x 上的一点且 q_i 到轨迹 R_x 的最短距离是已经计算过的。由于轨迹

R_x 和查询点 q_i 至少有一组已知的匹配点，则我们可以通过已知匹配点来计算出轨迹备选集中轨迹与查询点集相似度的下界，我们定义为 LB（lower bound）。

$$LB(R_x) = \sum_{1 \le i \le m \wedge R_x \in C_i} \left(\max_{1 \le j \le \lambda \wedge p_i^j \in R_x} e^{-D_e(q_i, p_i^j)} \right) \tag{3–6}$$

在计算下界的时候，我们考虑的查询点集为 $Q_{matched} = \{q_i | 1 \le i \le m \wedge R_x \in C_i\}$，对于 $Q_{matched}$ 中的查询点来说，他们和轨迹 R_x 上的某一个轨迹点在进行 k 最近邻查询中被计算作为匹配点，即对于查询点 $q_i \in Q_{matched}$，我们可以找到轨迹上的某一点 p_i^j 满足 $e^{-D_e(q_i, p_i^j)}$ 达到最大值，因为在欧氏距离上 q_i 与 p_i^j 最为接近，根据这一点我们可以将式3–6中的 $\max_{1 \le j \le \lambda \wedge p_i^j \in R_x} e^{-D_e(q_i, p_i^j)}$ 等价写成 $e_{-D_q(q_i, R_x)}$，得出式3–7

$$LB(R_x) = \sum_{1 \le i \le m \wedge R_x \in C_i} e^{-D_q(q_i, R_x)} \tag{3–7}$$

显然这个相似度下界值不会大于 $\sum_{i=1}^{m} e^{-D_q(q_i, R(x))}$，因为在计算相似度下界的时候值考虑了与在轨迹 R_x 上与某些查询点匹配轨迹点。另一方面，加入轨迹 R_x 不属于某个备选轨迹子集 C_i, 则轨迹 R_x 上任意轨迹点都不会存在于由查询点 q_i 得到的 k 最近邻查询结果 λNN 中。这说明由式3–6或3–7定义的相似度下界计算对于式3–2是成立的。

对于在不属于轨迹备选集 C 中的轨迹（$R_{ns} \notin C$），这些轨迹并没有在查询点集进行 k 最近邻查询中被扫描，他们到查询点 q_i 的距离将会小于 p_i^λ 到查询点 q_i 的最短距离，即 $D_e(q_i, p_i^\lambda)$。这一发现能让我们计算出所有不属于轨迹备选集的轨迹 R_{ns} 对于查询点集的相似度的上界，我们定义为定义为 UB_{ns}（upper bound）。

$$UB_{ns} = \sum_{i=1}^{m} e^{-D_e(q_i, p_i^\lambda)} \tag{3–8}$$

当我们在进行有关空间意义上的数据搜索的时候，剪枝是保证搜索效率的重要手段。相似度的上下界让我们可以设计出针对 k 最佳连接的剪枝算法以限制搜索空间，提高搜索效率，避免了对整个轨迹数据集或者对不满足条件的轨迹进行多余操作。根据相似度的上下界我们提出下述定理：

定理 3.1 (相似度上下界). 假设对于相似轨迹插叙的 k 最佳连接算法没有有序性限制，我们可以在对查询点集进行一轮 k 最近邻查询 (k=λ) 之后的轨迹备选集 C 中，选取一个包含 k 条轨迹的一个轨迹子集 C'。当 $\min_{R_x \in C'} LB(R_x) \ge UB_{us}$ 这一条件满足时，我们可以从轨迹备选集 C 中获得 k 条最佳连接轨迹，即 k 条与查询点集最相似的轨迹。

证明. 首先对于轨迹子集 C' 中的某一条轨迹 R_a（$R_a \in C'$）而言，轨迹 R_a 满足 $Sim(Q, R_a) \ge LB(R_a)$。与此同事，对于轨迹备选集 C 之外的轨迹 R_b（$R_b \notin C$），轨迹 R_b 满足 $UB_{ns} \ge Sim(Q, R_b)$。当上述定理成立时，即 $\min_{R_a \in C'} LB(R_a) \ge UB_{ns}$，我们可以推断出 $\forall R_a \forall R_b \big(R_a \in C' \wedge R_b \notin C\big)$，$Sim(Q, R_a) \ge Sim(Q, R_b)$ 成立。这也证明了对于查询点集 Q 得到的 k 最佳连接的结果轨迹在这个时候应该全部在轨迹备选集 C 中。 □

需要注意的是定理3.1中的轨迹子集 C' 不一定全是 k 最佳连接轨迹的结果，我们只能保证 k 最佳连接的轨迹在轨迹备选集 C 中。而定理3.1是我们在进行 k 最近邻查询而找到 k 最佳连接轨迹的保证条件。

3.2.3 增长型 k 最近邻查询算法

在搜索过程中我们需要解决一个关键问题就是 λ 的取值问题，λ 值的大小决定了我们的 k 最佳连接轨迹，即 k 最相似轨迹是否存在于轨迹备选集 C 中。在定理3.1中我们的轨迹备选集 C 的基数大小从在很大程度上取决于我们设定的查询 λ 值的大小。假如 λ 的值我们取的较大，则 k 最佳连接轨迹基本包含于轨迹备选集 C 内，但这样会造成搜索空间过大的问题。另一方面，λ 过小会使得轨迹备选集 C 不完全包含 k 最佳连接轨迹，导致搜索的不准确性和错误性。

算法 3–2 增长型 k 最近邻查询算法

输入: 相似轨迹查询数目 k, 查询点集 Q

输出: k 条最相似轨迹 $k\text{-}Trajs$

```
Candidate Set C; //初始化轨迹备选集 C
Initialise Upper bound of Similarity UB_ns; //初始化轨迹相似度上界
Initialise Lower bounds LB[], k − LB[]; //初始化两个有关相似度下界的数组
λ ← k //将 k 值初始赋值给 λ
while true do
    for each q_i ∈ Q that 1 ≤ i ≤ m do
        C_i ← trajectories that contains the points in λ-NN(q_i);
    end for
    C ← ⋃_{i=1}^{m} C_i; //合并轨迹备选子集以生成轨迹备选集 C
    if |C| ≥ k then
        compute LB[] for all trajectories in C; //计算轨迹备选集 C 中的所有轨迹相似度下界
        compute UB_ns; //计算相似度上界
        k-LB[] ← LB[].heapKtop(); //选取相似度下界 k 个最大值和其对应的轨迹
        if k-LB[].min ≥ UB_ns then //满足定理3.1
            k-Trajs ← refine(C) //轨迹筛选方法
            return k-Trajs
        end if
    end if
    λ ← λ + Δλ;
end while
```

为了解决 λ 的取值问题，我们尝试通过动态调整 λ 值的方法来一步步满足查询结果，这是我们提出增长型 k 最近邻查询算法的基本思想。增长性 k 最近邻查询算法是基于备选和筛选模式获取备

轨迹的高效算法，其大致思想是在算法开始对每一个查询点初步进行 k 最近邻查询（$k = \lambda$，最初 λ 可以是任一正整数值）。查询结果如果不满足定理3.1条件，我们再在 λ 值上进行动态增加 $\Delta\lambda$，将查询范围从 λ 增加到 $(\lambda + \Delta\lambda)$，然后我们进行 $(\lambda + \Delta\lambda)$ 最近邻查询。持续这一过程直到我们找到满足定理3.1条件的 λ 值。实现过程伪代码为算法3–2所示。

算法3–2的具体实现细节如下。通过函数主体首先定义初始化几个中间变量。$while$ 循环实现增长型 k 最邻近的每一轮查询。查询中，对查询点集 Q 中的每一个查询点进行 k 最近邻方法查询，对查询结果中的每一个轨迹点所在的轨迹都加入轨迹备选集 C。判断轨迹备选集 C 的基数大小是否满足条件。如果满足条件，计算此时归集备选集中所有轨迹的相似度下界大小并用一个数组 $LB[]$ 进行保存，同时计算未在轨迹备选集中的轨迹的相似度上界大小。运用堆排序或优先队列的思想将数组 $LB[]$ 中选取 k 个最大相似度下界及相对应的轨迹。如果3.1满足，即选出来 k 条轨迹中的最小轨迹相似度下界大于等于未在轨迹备选集中的轨迹相似度上界，则说明我们需要查询的 k 条最相似轨迹已经存在于轨迹备选集 C 中，并且其他未扫描到的轨迹可以忽略不予以计算与检查。

3.2.4 轨迹筛选算法

在增长型 k 最近邻算法实现过程中，我们利用数组 $LB[]$ 进行堆排序或优先队列获取的 k 条轨迹相似度下界最大的对应轨迹并不能直接作为我们所需要查询的 k 条最相似轨迹。第一，轨迹相似度下界并不能直接作为评判轨迹之间相似程度的大小的比较标注；第二，有可能存在多于 k 条轨迹，他们的相似度下界均大于此时的未在轨迹备选集中的轨迹相似度上界。因此我们需要在增长型 k 最近邻算法中加入轨迹筛选算法，找到真正满足轨迹相似的 k 条轨迹。

在增长型 k 最邻近算法中，我们通过轨迹筛选算法 $refine(C)$，对轨迹备选集 C 中的轨迹进行剪枝，去掉不符合条件的轨迹然后再获取 k 条最相似轨迹。实现轨迹筛选算法，我们仅针对已有的轨迹备选集 C 定义其中轨迹的相似性上界。

$$UB_(R_x) = \sum_{1\le i\le m \wedge R_x \in C_i} (\max_{1\le j\le \lambda \wedge p_i^j \in R_x} \{e^{-D_e(q_i,p_i^j)}\}) + \sum_{1\le i\le m \wedge R_x \notin C_i} (e^{-D_e(q_i,p_i^\lambda)}) \tag{3–9}$$

式3–9中，轨迹 $R_x \in C$。对于每一个满足条件 $1 \le i \le m \wedge R_x \in C_i$ 的查询点 q_i 而言，轨迹 R_x 到查询点 q_i 的的最短距离是在进行 k 最近邻查询 λ-NN(q_i) 中被计算过。因此我们通过这些最短距离匹配点对 $< q_i, p_{closest} >, p_{closest} \in R_x$ 来计算轨迹备选集中轨迹的相似度上界。对于 k 最近邻查询结果不包含轨迹 R_x 上任意一点的查询点 q_j 而言，我们考虑 q_j 进行 k 最近邻查询 λ-NN(q_i) 的第 λ 个点，即 p_j^λ。$< q_j, p_j^\lambda >$ 之间的距离肯定比 $D_q(q_j, R_x)$ 要近，因此在计算备选集中的轨迹的相似度上界时选择使用 $D_e(q_j, p_j^\lambda)$。

$$\begin{aligned}\forall R_x \in C, Sim(Q, R_x) &= \sum_{i=1}^{m}(e^{D_q(q_i,R_x)}) - \sum_{1\le i\le m \wedge R_x \in C_i}(e^{-D_q(q_i,R_x)}) - \sum_{1\le i\le m \wedge R_x \notin C_i}(e^{-D_e(q_i,p_i^{\lambda})}) \\ &= \sum_{1\le i\le m \wedge R_x \notin C_i}(e^{D_q(q_i,R_x)} - e^{-D_e(q_i,p_i^{\lambda})}) \\ &\le 0\end{aligned} \tag{3–10}$$

式3–10证明了对于轨迹备选集 C 中的任意一条轨迹而言，UB 是轨迹相似性的上界值，即 $\forall R_x \in C, Sim(Q, R_x) \le UB(R_x)$。

算法3–3为轨迹筛选算法的实现细节。首先运用优先队列思路，维护一个目前为止的 k 条最相似轨迹的数组或者队列，并暂时保存其对应的相似度。对轨迹备选集 C 中的每一条轨迹计算其对应的轨迹相似度上界并以此为关键字对轨迹进行从大到小的排序。筛选轨迹算法终止条件当且仅当目前轨迹数组或队列中的最小轨迹相似度大于还未进入过队列的轨迹相似度上界的最大值，此时 k 条最相似轨迹被准确找出。在循环体中，我们维护轨迹数组或轨迹队列，并在找到一条更匹配或更相似与查询点集的轨迹时更新我们已有的数组和队列。

算法 3–3 轨迹筛选算法 refine(C)

输入： 相似轨迹查询数目 k, 轨迹备选集 C

输出： k 条最相似轨迹 $k\text{-}Trajs$

```
Initialise k-Trajs as an array or a priority queue
Compute the Upper bound of Similarity, UB, for each trajectory in candidate C
Sort trajectory in candidate C by UB in descending order
for x in range(1, |C| + 1) do
    compute Sim(Q, R_x);
    if x ≤ k then
        k-Trajs.insert(R_x, Sim(Q, R_x));
    else
        if Sim(Q, R_x) > k-Trajs.minSim then
            k-Trajs.removeMinSimTrajectory();
            k-Trajs.insert(R_x, Sim(Q, R_x));
        end if
        if x = |C| + 1 or k-Trajs.minSim ≥ UB(R_{x+1} then
            return k-Trajs;
        end if
    end if
end for
```

3.3 算法优化

3.3.1 λ 动态增长优化

在增长型 k 最近邻查询算法3–2中，对于每一次的 k 最近邻查询 λ-NN(q_i) 而言，搜索范围 λ 都是动态增加 $\Delta\lambda$，即每一轮循环中，对于查询点集中的每一个查询点 q_i，搜索范围在数目上是相等的。但值得提出的时，在针对地理位置点进行相似轨迹查询这一上下文中，查询点对于结果的重要性并不是完全一致的。主观而言，有些位置点相对于其他位置点来说具有更重要或更优先的查询级别；从算法角度讨论，每个查询点 q 的结果 λ-NN(q) 对于构建轨迹备选集 C、决定轨迹相似度上下界均有着不同的影响程度。举例来说，对于两个查询点 q_i 和 q_j，在 λ 相同的情况下，如果 $D_e(q_i, p_i^\lambda) > D_e(q_j, p_j^\lambda)$，则对于查询点 q_i 所查找的范围更大，即 $e^{-D_e(q_i,p_i^\lambda)} < e^{D_e(q_j,p_j^\lambda)}$。在式3–8中，$UB_{ns} = \sum_{i=1}^{m} e^{-D_e(q_i,p_i^\lambda)}$，我们可以根据结果推出在降低未在备选集中的轨迹相似度上界的过程中，查询点 q_i 比查询点 q_j 效果更好，更有帮助。在定理 3.1.2. 中，未在备选集中的轨迹相似度上界越低，则定理条件越容易满足，即增长型 k 最近邻查询算法可以更快得出结果。我们需要分析 λ 对每个查询点搜索的影响来决定如何动态增加搜索范围。首先我们定义每个查询点 q_i 对于 UB_{ns} 的影响为 $\xi(q_i)$

$$\xi(q_i) = e^{-D_e(q_i,p_i^\lambda)}$$

显然，当 $\xi(q_i)$ 的值越小时，则相对应的 UB_{ns} 也将越小。接着我们定义 ρ 为某一范围内轨迹点的密度值，定义 $r = D_e(q_i, p_i^\lambda)$ 为对查询点 q_i 进行 k 最近邻查询时的搜索半径。在 k 最近邻查询这一范围内，我们可以粗略计算出轨迹点的密度值 ρ 等于

$$\rho = \frac{\lambda}{\pi r^2}$$

根据轨迹点密度和搜索半径的关系，我们重写 $\xi(q_i)$ 为

$$\xi(q_i) = e^{-D_e(q_i,p_i^\lambda)} = e^{-r} = e^{-\sqrt{\frac{\lambda}{\pi\rho}}}$$

在这一步，我们的首要目标是明确 $\xi(q_i)$ 影响因子的下降速度与 λ 之间的关系，根据 λ 的变化所造成的影响赋予查询点 q_1 到 q_m 不同的 $\Delta\lambda$ 变化值，即对于不同的查询点，除了初始第一轮查询之外，之后 ($\lambda + \Delta\lambda$) 的值都是各自生成的。本文将 $\xi(q_i)$ 为关于 λ 的微分值 $\frac{d\xi}{d\lambda}$ 的绝对值定义为下降速率 $Decay(q_i)$

$$\frac{d\xi}{d\lambda} = \frac{d}{d\lambda} e^{-\sqrt{\frac{\lambda}{\pi\rho}}} = -\frac{1}{2}(\pi\rho\lambda)^{-\frac{1}{2}} * e^{-\sqrt{\frac{\lambda}{\pi\rho}}} \tag{3–11}$$

根据式3–11，我们可以用 λ 和搜索半径 r 来计算轨迹点密度 ρ，因此可以改写下降速率为

$$Decay(q_i) = |\frac{d\xi}{d\lambda}| = \frac{r}{2\lambda} e^{-r} \tag{3–12}$$

根据式3–12，我们可以得知，对于一个固定的 λ 值来说，下降速率 $Decay(q_i)$ 会随着搜索半径 r 的不断增长，先初步上升（$r \in (0, 1]$）后逐渐下降 $r \in (1, \infty)$）。我们可以得知在对于查询结果较为稀疏的查询点（即搜索半径 r 较大）在一开始赋予较大的查询权重值。但随着搜过过程的进行，当搜索半径 r 不断增长达到某一个值得时候，一些相对密集的查询点结果会使得其对应的下降速率变

大。这一结论使得我们在搜索和查询过程中重点关注查询点结果较为密集的查询点，这样也能使得我们能更快更有效地在每一轮查询之后降低未在轨迹备选集中轨迹的相似度上界值 UB_{ns}。但随之产生的问题在于，当搜索半径 r 和 λ 都足够大的时候，我们下降 UB_{ns} 会因为 $\frac{ds}{d\lambda}$ 趋近于 0 而变得不再有效。

满足定理3.1需要上下界两个变量对条件的同时满足。因此，在关注未在轨迹备选集中轨迹的相似度上界值 UB_{ns} 对增长型 k 最近邻查询的影响时，我们可以在加速增长型 k 最近邻查询算法的时候考虑相似度下界这一因素。当备选集中轨迹的相似度下界 LB 增长越快的时候，定理3.1也就越容易成立。提高相似度下界 LB 所要面对的问题在于，一条轨迹的相似度下界有可能是源于多个查询点所产生查询结果，并且想要预测在搜索过程中什么时候 λ-NN(q_i) 的结果中的某一点和轨迹上的某一点恰好是同一个点也是不太容易的。换言之，我们问题主要在于定量描述每一个查询点对于相似度下界增长的影响。借此，我们基于每一轮重新查找到的新轨迹数目来定义一个启发式搜索的取回速率 $Ratio(q_i)$

$$Ratio(q_i) = \frac{Number(q_i)}{\Delta\lambda} \tag{3–13}$$

式3–13中 $\Delta\lambda$ 为当前循环轮次 λ 的值与上一轮循环中 λ' 值的差值（$\lambda > \lambda'$），而 $Number(q_i)$ 表示在当前循环轮次搜索中获取的轨迹数目多少。基本思想在于，轨迹备选集 C 的基数值范会随着搜索过程中新轨迹数目的增长而增长。在这样的归集备选集 C 中，轨迹相似度下界会曾铮的更快，再根据定3.1，我们也更有可能找到目标寻求的 k 条最相似轨迹。

结合考虑上文所提及的下降速率 $Decay(q_i)$ 和取回速率 $Ratio(q_i)$，我们可以对每一个查询点指定对应的 λ 查询增长值 $\Delta\lambda(q_i)$

$$\Delta\lambda(q_i) = \gamma\big(\alpha\frac{Decay(q_i)}{\sum_{i=1}^{m} Decay(q_i)} + \beta\frac{Ratio(q_i)}{\sum_{i=1}^{m} Ratio(q_i)}\big) \tag{3–14}$$

式3–14中，α 和 β 是本文定义的权值，γ 定义为 $\gamma = mk2^r$ 其中 r 为算法增长型 k 最近邻查询的当前循环轮次数。这样，我们摈弃原先对每一个查询点都增长相同的 λ 值这一处理思路，选择通过式3–14的方法应用于每一个查询点上以对每个查询的进行不同的 λ 增量处理。这样的预先处理会在挖掘出相对重要的轨迹查询点上花费一定时间，但也加速了整个增长型 k 最近邻查询算法的搜索过程。这样的预处理时间由于优化整个算法过程，因此是可接受的。注意到我们在每一轮 λ 增量的总值是

$$\sum_{i=1}^{m}\Delta\lambda(q_i) = \gamma\big(\alpha\frac{\sum_{i=1}^{m} Decay(q_i)}{\sum_{i=1}^{m} Decay(q_i)} + \beta\frac{\sum_{i=1}^{m} Ratio(q_i)}{\sum_{i=1}^{m} Ratio(q_i)}\big) = \gamma(\alpha+\beta)$$

为了保证在每一轮增长型 k 最近邻查询过程中获取的结果轨迹点数据恒定，我们将 $\alpha+\beta$ 设定为 1，其中可以设定 $\alpha = \beta = 0.5$，这样每一轮我们获取的点的数据为 γ

3.3.2 基于动态规划实现有序查询

在前文中我们提及查询的有序性和用户指定有关。在进行有序查询的过程中，之前的算法是基于递归进行实现的：通过去不断匹配轨迹和查询点来进行子递归，从而计算出轨迹和有序查询点集

之间的相似度大小。但基于递归相似度计算会占用大量的时间。因此在本上，我们通过动态规划的思路来计算某一条轨迹 R 和查询点集 Q 的相似度，借此来优化算法在有序查询中的处理性能。

具体而言，我们借助3–4算法来处理查询中的有序相似度计算。这里算法的输入为查询点集 $Q = q_1, q_2, q_3, \cdots, q_m$ 和轨迹 $R = p_1, p_2, p_3, \cdots, p_n$，并通过不断重复或略过轨迹上的点 p_j 来得到最好的匹配结果以计算有序相似度。

算法 3–4 有序相似度算法 dp_Similarity(Q,R)

输入: 查询点集 $Q = q_1, q_2, q_3, \cdots, q_m$, 轨迹 $R = p_1, p_2, p_3, \cdots, p_n$

输出: 查询点集 Q 和轨迹 R 之间的有序相似度 $Sim_{order}(Q, R)$

```
Initialise 2-dimensional array M[m+1][n+1];
M[i][0] ← 0 for 1 ≤ i ≤ m;
M[0][j] ← 0 for 1 ≤ j ≤ m;
for 1 ≤ i ≤ m do
    for 1 ≤ j ≤ n do
        if e^{-D_e(Head(Q),Head(R))} + M[i-1][j] > M[i][j-1] then//q_i 和 p_j 匹配
            M[i][j] ← e^{-D_e(Head(Q),Head(R))} + M[i-1][j]; //重复 p_j
        else
            M[i][j] ← M[i][j-1]; //略过 p_j
        end if
    end for
    return M[m][n];
end for
```

算法3–4中，$M[i][j]$ 是我们需要解决查询问题的子问题的有序相似度，即 $Sim_{order}(\{q_1, q_2, q_3, \cdots, q_i\}, \{p_1, p_2, p_3, \cdots$
对于动态规划思路而言，当我们获取到 $M[i-1][j]$ 和 $M[i][j-1]$ 的值时，我们可以通过比较 $e^{-D_e(Head(Q),Head(R))}+M[i-1][j]$ 和 $M[i][j-1]$ 的值来决定 $M[i][j]$ 的最大值。如果值 $e^{-D_e(Head(Q),Head(R))}+M[i-1][j]$ 较大，我们可以得出目前的一对匹配点对为 $< p_i, p_j >$，并令 $M[i][j] = e^{-D_e(Head(Q),Head(R))}+M[i-1][j]$，反之，我们略过对 p_j 的目前和之后匹配，并令 $M[i][j] = M[i][j-1]$。这一动态规划的思路自底向上的解决了 $M[i][j]$ 的求值问题，其中 m 为查询点集的基数大小而 n 为轨迹点数目。在算法最后通过范围二维数组中的值来表示查询点集 Q 和轨迹 R 之前有序相似度。算法的复杂性为 $O(mn)$，在具体应用中由于 m 的值相对于 n 来说普遍较小，所以我们可以将算法复杂性近似看成是线性的。

之后的问题在于如何将有序相似度与增长型 k 最近邻查询算法结合。首先，对于轨迹备选集 C 中的一条轨迹 R 而言，$R \in C$，轨迹 R 中的某些轨迹点存在于增长型 k 最近邻查询过程中的某一个或者几个 λ-NN 结果中，我们定义这些轨迹点为 R'，显然 $R' \subseteq R$，即

$$R' = \{p_i | p_i \in R \wedge p_i \in \bigcup_{j=1}^{m} \lambda - NN(q_j)\}$$

基于有序查询，我们可以得出

$$Sim_{order}(Q,R) \geq Sim_{order}(Q,R') \tag{3–15}$$

证明式3–15可以依据反证法：假设 $Sim_{order}(Q,R) < Sim_{order}(Q,R')$，其中 Q 和 R' 之间的匹配点对位 $\{< q_1, p(\varphi_1) >, < q_2, p(\varphi_2) >, \cdots, < q_m, p(\varphi_m) >\}$，其中 $p(\varphi_i) \in R'$，根据有序相似度的定义，$Sim_{order}(Q,R') = \sum_{i=1}^{m} e^{-D_e(q_i,p(\varphi_i))}$，由于 $R' \subseteq R$，即有 $p(\varphi_i) \in R$，那么 $sum_{i=1}^{m} e^{-D_e(q_i,p(\varphi_i))}$ 对于 $Sim_{order}(Q,R)$ 也是成立的，则 $Sim_{order}(Q,R)$ 至少大于等于 $Sim_{order}(Q,R')$，与假设矛盾。

因此，我们重新修改我们相似性的上下界以使他们使用于有序查询的情况。根据式3–15，我们将相似性的下界通过 R' 上所获取的点来定义；对于相似性的上界我们仅针对轨迹备选集 C 中的轨迹 R_c $R_c \in C$ 来定义

$$\begin{aligned} LB_{order}(R) &= Sim_{order}(Q,R') = dp\_Similarity(Q,R') \\ UB_{order}(R_c) &= LB_{order}(R_c) + \sum_{1 \leq i \leq m \wedge R_c \notin C_i} \left(e^{-D_e(q_i,p_i^{\lambda})}\right) \end{aligned} \tag{3–16}$$

根据式3–16将相似度上下界从无序的定义改成有序的定义，则可以在增长型 k 最近邻查询中加入有序查询限制，即根据查询点集的查询顺序查询最相似的 k 的轨迹。

第四章　基于分布式处理相似轨迹查询

4.1　大规模数据集群处理

集群计算随着如今海量数据的发展在许多领域都有着广泛应用，以高效准确完成大量数据并行级的处理任务。集群计算需要提供本地化任务规划、高容错机制和数据负载平衡基本功能。除此之外，集群计算中我们也会关注某一个数据集运用在并行操作去完成指定目标任务。*MapReduce* 是目前分布式处理或并行计算常用的大规模数据处理和生成的编程模型。其工作的大致思路在于用户自定义合适的 map 函数去处理初步输入的键值对数据并产生中间键值对结果，之后定义 $reduce$ 函数将中间结果以相同的关键字进行合并生成最终的结果。

对于数据密集型的应用而言，可扩展的分布式系统对于系统运行和数据处理都有着很重要的帮助。合理的分布式系统可以为系统提供在通用硬件上运行时的容错保护，并且能保证多用户请求的高度并行处理。*Hadoop* 分布式文件系统（*HDFS*）借鉴了 *Google* 文件系统（*GFS*）的大部分设计架构并实现了高度的容错保护机制并且能良好地运行于廉价的硬件设备之上。与此同时，HDFS 也保证了在应用中数据的高度吞吐速率，使得 HDFS 能高效运行具有很大数据集的任务或应用。

4.1.1　Spark 处理引擎简介

MapReduce 变成模型和 *HDFS* 可在大规模数据密集型应用良好，但对于一些需要重复使用中间数据或需要暂时保留中间数据的应用处理中，之前常用的集群计算模型 *Hadoop* 由于需要将中间数据读写与 *HDFS* 中从而产生了中间读写时间浪费，从而影响了应用性能。基于这一点，*Spark* 作为在主流针对大规模数据处理的集群计算模型之一，在保证之前集群计算模型功能的同时，使用一种名为弹性分布式数据集（*Resilient Distributed Datasets*）的抽象，使得其可以将集群任务中的中间结果保存于设备的内存之中，以便之后的读写操作。因此，在大数据挖掘领域中，*Spark* 能够比 *Hadoop MapReduce* 能为高效的处理需要迭代数据的集群计算。

4.1.2　弹性分布数据集 RDD

Spark 集群计算处理引擎与之前集群处理的主要不同点即在于其引入的弹性分布式数据集（*RDD*）这一抽象概念。这一分布式内存抽象使得用户或程序员可以在容错机制的保护下在大规模集群设备中运行基于内存的数据操作。*RDD* 高效处理大数据在应用中的重用问题，作为一个并行的数据结构可以方便用户在内存中处理集群计算的中间过程数据，因地制宜分割数据集以更合理将任务分配个对应的工作节点，再结合丰富的内定操作函数以快速处理数据。而 *RDD* 提供的生成模式也能为我们设计算法提供更多思路。*RDD* 可以通过 $parallelize$ 函数将程序中已有的数据用于生成为 *RDD*，或通过对例如 *HDFS*、*Hbase* 等等的外部文件系统或外部数据源来生成 *RDD*。

4.1.3 Spark Standalone 集群模式

Spark 应用在集群模式运营中运行独立的进程组，通过驱动程序中的 *Spark* 上下文变量 *SparkContext* 来设定运行参数和初始化。运行过程主要根据 *SparkContext* 来连接如图4–1中具体不同种类的集群管理类型，并通过内定的 *Cluster Manager* 来分配应用的资源获取。初始化成功后，*Spark* 通过获取集群节点上的执行进程并准备开始执行操作和处理数据。之后，*Spark* 会将应用代码分发给各个节点并使之运行。

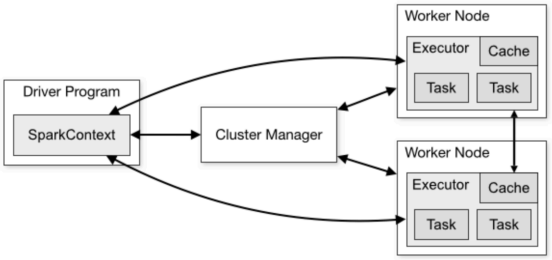

图 4–1 集群管理大致模式

Fig 4–1 Cluster Managing Mode

在相似轨迹获取这一应用中，根据我们的应用场景和硬件设置，我们采用 *Spark* 自带的 *Standalone* 集群模式，其大致设计框架如图4–2所示。

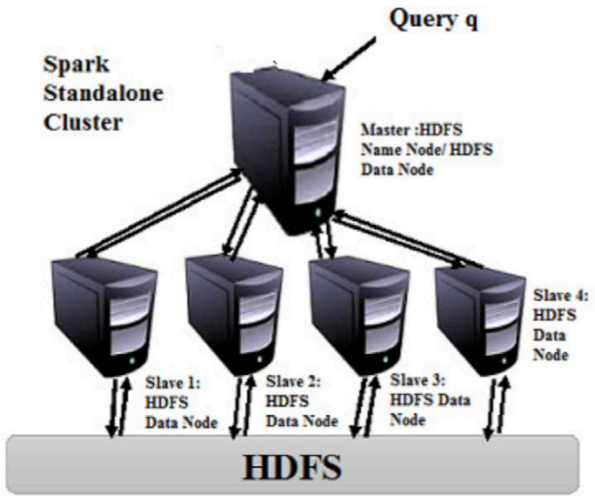

图 4–2 Spark Standalone 集群结构

Fig 4–2 Spark Standalone Architecture

在 *Standalone* 集群模式中，我们通过一个在 *Spark* 分布式环境下的简要集群管理者来简单建立集群处理环境。在这一集群环境中，主节点（*Master Node*）为驱动程序运行的设备节点。驱动程序不仅是与用户交互信息的接口（*interface*）程序，也负责分布式运行在 *Spark* 环境中进程的运行情况。子节点们（*Slave Nodes*）为启动在工作节点中的进程提供运行环境，这些进程运行任务代码并在内

存或磁盘中储存数据。在相似轨迹搜索中，我们将轨迹数据和预处理好的轨迹索引 R 树结构存储在 *HDFS* 上，集群环境中的工作节点可以通过设定好的参数无需秘钥的共享 *HDFS* 上已存储好的数据，这样，我们可以将相似轨迹搜索任务进一步以分布式的方法进行处理。

4.2 分布式相似轨迹查询算法实现

在之前的工作描述中，我们对于相似轨迹查询的实现总会提及查询点集的数目相对较少这一前提。对于单机处理而言，如果将一整条轨迹的轨迹数据点作为输入或者查询点集过多，由于增长型 k 最近邻查询会对每一个查询点都进行 λ-NN 的搜索处理，因此整个相似轨迹的查询过程会显得相对缓慢。但有些时候，将一条轨迹作为相似轨迹查询的输入的确更简单且更人性化。在硬件设置对算法性能的约束下，借助基于地理位置点的轨迹简化方法，我们可以对前一章所涉及的相似轨迹查询方法通过加入 *Spark* 分布式集群处理的手段，做到以一条轨迹（或数量更多的查询点）为输入的相似轨迹查询操作。具体实现思路在于，通过基于地理位置点的轨迹简化方法将一条轨迹简化为一组数量相对于单机查询要多的查询点集；然后将查询点通过分布式集群操作分配给各个子节点来进行相对于各集群节点的相似轨迹查询操作，各个子节点通过访问 HDFS 获取轨迹数据和轨迹 R 树索引；最后将结果以轨迹为关键字，以相似度为权值进行求和，选择相似度和最高的 k 条轨迹作为查询结果。

4.2.1 基于地理位置点的轨迹简化方法

前文描述的 $Douglas–Peucker$ 轨迹简化算法主要是在保留大致形状信息的基础上对减少一些距离较近的轨迹点，从而做到轨迹简化。但对于一条轨迹而言，轨迹除了轨迹点数目这一关键字，其中一些轨迹点也具有一些隐藏的轨迹语义信息。在现实生活中，一些岔路口可能在一条轨迹中称为比较关键和重要点，或者某一个立交交通系统会成为路线中转的关键轨迹点，因此在做轨迹简化中应该尽可能地保留这些点。

在本文中，主要采用基于地理位置点的轨迹简化算法，称为 $TS\ algorithm$（TS），借鉴语义的重要性，它主要思路是从一条轨迹中查找重要的或具有特定语义的轨迹点从而进行简化和压缩。TS 轨迹简化算法既保证了轨迹的大致路线也保留了重要的特定轨迹点。基于轨迹点的相似轨迹查询用户大概率会使用一些在地理意义上重要的轨迹点，因此选择基于地理语义的轨迹压缩方法能够将轨迹简化结果和相似轨迹查询匹配使用。*TS* 轨迹简化方法中，每个轨迹点的前进角度大小（*heading direction degree*）和其余相邻轨迹点之间的距离作为衡量该轨迹点权值的主要因素。通过正则化垂直距离对轨迹点权值的影响，*TS* 轨迹简化方法在效果上要优于 $Douglas–Peucker$ 算法。

定义 4.2.1. 前进角度（*Heading direction*）h 以正北方向为基准，表示一个移动物体之后的行进方向，其中 $0° \leq h < 360°$

定义 4.2.2. 近邻前进角度改变量（*Neighbor Heading Change*）α 是对于某一个轨迹点 p_i 而言，其前进角度 $p_i.h$ 和之前一点 p_{i-1} 的前进角度 $p_{i-1}.h$ 的差值，即 $p_i.h - p_{i-1}.h$，其中其中 $-180° < \alpha < 180°$

定义 4.2.3. 前进角度积累量（*Accumulated Heading Change*）β 表示轨迹点 p_i 前后范围 ϕ 内的近邻前进角度改变量之和，即 $p_i.\beta = \sum_{k=i-\phi}^{i+\phi} p_k.\alpha$。其中的 ϕ 为认为预先定义好的整数阈值。

定义 4.2.4. 前进量改变值（*Heading Change*）γ 用近邻前进角度改变量 α 和前进角度积累量 β 定义，前进量改变值为后面两者绝对值之和，即有 $p.\gamma = |p.\alpha| + |p.\beta|$

定义 4.2.5. 近邻距离（*Neighbor Distance*）是指一个轨迹点 p_i 到前一个轨迹点 p_{i-1} 和后一个轨迹点 p_{i+1}（如果存在的话）的欧式距离之和，我们用符号 d 代表这一距离和，即 $p_i.d = Dist_e(p_{i-1}, p_i) + Dist_e(p_i, p_{i+1})$，其中，$Dist_e$ 在前文中已有定义，不再赘述。

定义 4.2.6. 轨迹点权值（*Weight*）表明轨迹点 p 的在轨迹简化中重要性大小，通过前进量改变值和近邻距离定义，即 $p.\omega = p.d * p.\gamma$。

根据上述定义，我们可以设计出基于地理位置点的轨迹点简化算法4–5。该算法主要分为四个过程：分段、段权值排序、点权值计算和点选择。由于本工作数据集只针对车载轨迹数据，则在第一步只用进行简单的分段操作而不用进行轨迹类型分类操作。而第二步根据设定参数求出每一段的权值并通过算法定义参数保留段轨迹语义。第三步计算每个数据点的权值并通过正则化决定每一段子轨迹内选择保留相对比例的数据点。最后选择符合算法条件的点作为简化结果返回。

算法 4–5 轨迹简化（Trajectory Simplification）算法

输入: 一条原始轨迹数据 $Traj$, 简化后轨迹点数目 m

输出: 一条简化后只有 m 个点的轨迹 $Traj'$

```
1: Traj' ← ∅
2: Seg[] ← Segmentation(Traj) //将轨迹 Traj 分段
3: DistributePoints(Seg[], m); //求段权值并参数初始化
4: for Each Segment s in Seg[] do
5:     WeightPoints(s) //求点权值
6:     s' ← SelectPoints(s) //选择点组成新的轨迹段
7:     Traj' ← Traj ∪ s' //合并新轨迹段组成结果
8: end for
```

图4–3展示了一条原始数据轨迹 $Traj$ 通过 *TS* 轨迹简化算法后的结果。

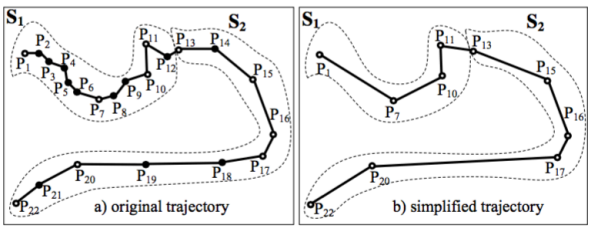

图 4–3 *TS* 算法结果样例

Fig 4–3 An illustration of Trajectory Simplification Algorithm

4.2.2 Spark 分布式相似轨迹查询

Spark 集群环境使得我们可以将代码分发给各个工作节点使他们处理对数据集相同的操作，这为分布式搜索相似轨迹提供了实现的基础。单机实现相似轨迹搜索为保证运行性能，给定的输入集查询点需要保证数量在某一程度上相对较小。但对于分布式处理而言，这一约束可以通过集群集计算处理予以取消。给定一条原始轨迹 $Traj$，我们可以先通过基于地理位置点的轨迹简化在保留轨迹形状和轨迹中的重要位置点的同时，一定程度上减少轨迹数目。事实上，如果集群设备性能较好，我们可以略去集群计算相似轨迹前对轨迹简化这一步骤。由于本文实验设备限制，通过在集群处理前的轨迹简化能在保证结果正确性的过程中，稳定处理性能。因此，我们将轨迹简化作为集群计算前的预处理过程。

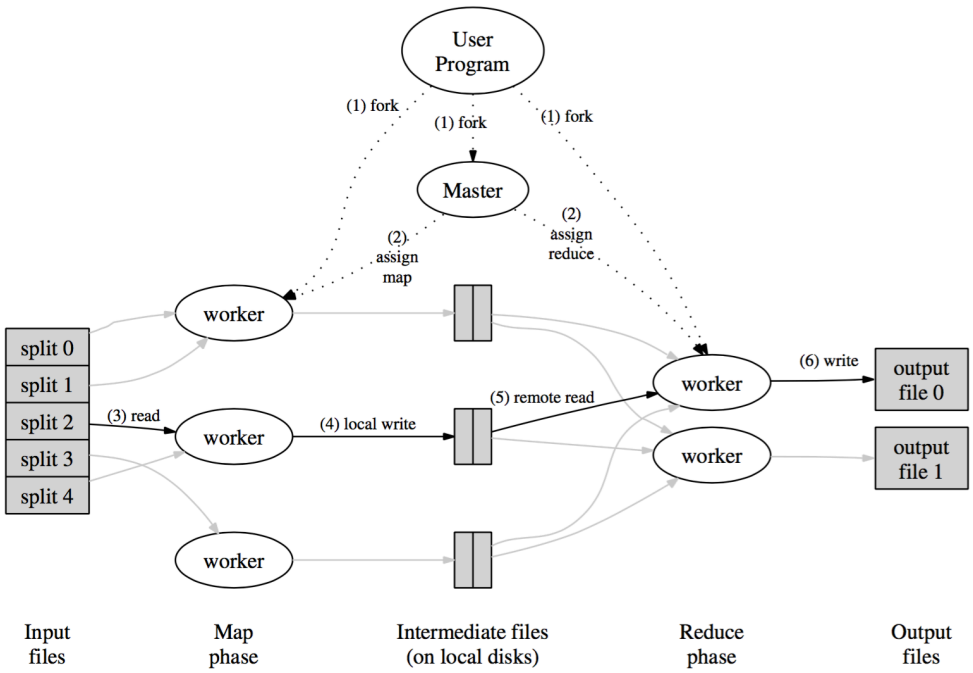

图 4–4 Spark 分布式相似轨迹查询

Fig 4–4 Spark distributed search similar trajectories

图4–4为 *MapReduce* 的大致处理框架，我们根据这一框架设计出分布式查询相似轨迹的大致算法。在算法中，假设我们已经完成对原始轨迹的简化得到已处理好的简化轨迹 $Traj'$。首先我们通过 *Spark* 内的 *partition* 函数将输入数据分割，并分发给每一个工作节点（*Worker Node*）。对于每一个工作节点而言，他们都有了输入数据的一部分轨迹点。对于每一个工作节点而言，他们都可以将自己所拥有的部分轨迹点作为相似轨迹查询的查询点集，由于工作节点通过设置可以直接访问 *HDFS* 上已存储的轨迹数据，我们通过主节点（*Master Node*）将增长型 k 最近邻查询代码分发给每一个工作节点并予以上运行。每个工作节点将针对各自输入查询点集所得出的 k 条最相似轨迹及其对应的相似度大小作为输出。通过对轨迹作为关键字合并中间查询结果，对相似度求和按相似度大小降序排列轨迹，再从中选取 k 条轨迹作为最终的结果。

实现细节如算法4–6所示。其中在运行过程中初始化 *Spark* 运行所需的上下文变量并设置主节点信息。在 *map* 阶段对查询点集进行查询搜索，在 *reduce* 阶段进行关键字合并。

算法 4–6 分布式相似轨迹查询算法

输入: 相似轨迹查询数目 k, 一条原始轨迹数据 $Traj$

输出: k 条最相似轨迹 $k\text{-}Trajs$

```
1: Traj' ← TS(Traj);
2: Initialise Spark Context sc and set Master information;
3: RDD ← sc.parallelize(Traj', partitionNumber);
4: res ← RDD.map(iknn)        //工作节点分布式进行查询处理
5:         .flatMap(lambdax:x)        //查询结果平铺成为一维列表或数组
6:         .reduceByKey(lambdax, y:x + y)        //以轨迹为关键字做相似度求和
7:         .sortBy(lambdax:x[1], descending)        //降序排列
8:         .collect();
9: return res;
```

4.2.3 多请求分布式相似轨迹查询

致　谢

感谢所有测试和使用交大学位论文 LaTeX 模板的同学!
感谢那位最先制作出博士学位论文 LaTeX 模板的交大物理系同学!
感谢 William Wang 同学对模板移植做出的巨大贡献!