**Graphhopper——Map-matching**

# 项目结构简介

* Hmm-lib维比特算法
* Map-data地图数据

\*.osm.pbf

\*.osm.gz

* Matching-core核心类

MapMatchingTest.java

void testDoWork()

* Matching-web 网页应用

/src/main/resources/asset

->index.html

* Matching-web-bundle 后台处理matching-web的index.html的事件

MapMatchingBundle.java

MapMatchingResource.java

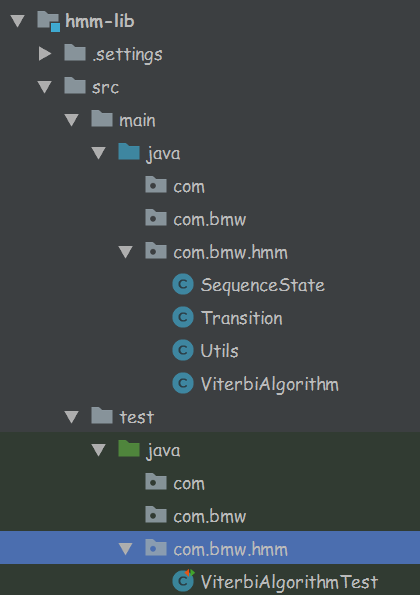
引用Matching-core的部分类

# 项目结构及API的详细介绍

## Hmm-lib

这个库实现了用于时间非均匀马尔可夫过程的隐马尔可夫模型（HMM）。这意味着，与许多其他HMM实现相反，在每个时间步骤可以存在不同的状态和不同的转换矩阵。//转移矩阵：正态分布

<https://www.cnblogs.com/mindpuzzle/p/3653043.html>



### Test：测试代码

\* @param <S> the state type —— Rain

\* @param <O> the observation type —— Umbrella

\* @param <D> the transition descriptor type —— Descriptor

#### Hmm—ViterbiAlgorithm算法

<http://www.hankcs.com/nlp/hmm-and-segmentation-tagging-named-entity-recognition.html>

<http://www.hankcs.com/nlp/general-java-implementation-of-the-viterbi-algorithm.html>

#### 一些假设

假设一：

下雨带伞的概率为0.9

不下雨带伞的概率为0.2

下雨不带伞的概率为0.1

不下雨不带伞的概率为0.8

假设二：

下雨->下雨的概率为0.7

下雨->不下雨的概率为0.3

不下雨->下雨的概率为0.3

不下雨->不下雨的概率为0.7

#### 已知条件

带伞->带伞->不带伞->带伞

#### 推出概率

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 下雨 | 0.9 | 0.567=0.7\*0.9\*0.9 | 0.03969=0.567\*0.1\*0.7 | 0.0367416=0.13608\*0.3\*0.9 |
| 不下雨 | 0.2 | 0.054=0.2\*0.3\*0.9 | 0.13608=0.567\*0.3\*0.8 | 0.0190512=0.13608\*0.2\*0.7 |

#### 问题迁移：Map-Matching的两个变量：

**HMM显状态：**从GPS设备中得到的位置信息（经度，纬度）：

**HMM隐状态：**拥有GPS设备的物体（车，人等）实际所在的位置变量，实际地图是不知道GPS设备的准确位置的。

**观测概率：**观测的GPS点离旁边路段上的位置越近，那么这个真实点在这个路段上的概率越大

**状态转移概率：**这里有两种解决思路：

（1）前后两个真实的位置点的距离越近，那么状态转移的概率越大；

（2）真实路段上的前后两个点的距离与GPS观测的前后两个点的距离越接近，状态转移概率越大。

### Main

#### SequenceState<S, O, D>

public final S state;//隐状态

public final O observation;//显状态

public final D transitionDescriptor;//转换描述

#### Transition<S>

public final S fromCandidate;//上一个状态

public final S toCandidate;//下一个状态

#### Util

Int initialHashMapCapacity();

//用于确定hashmap的大小，ViterbiAlgorithm中调用

#### ViterbiAlgorithm<S, O, D>//主要算法的类

## Map-data

\*.osm.pbf

\*.osm.gz

## Matching-core

Graph-cache：编译后产生的文件夹，缓存部分地图数据。

文件夹src/main/resources/asset/mapmatching-webapp下：

## Matching-web

-> index.html

->js

->demo.js

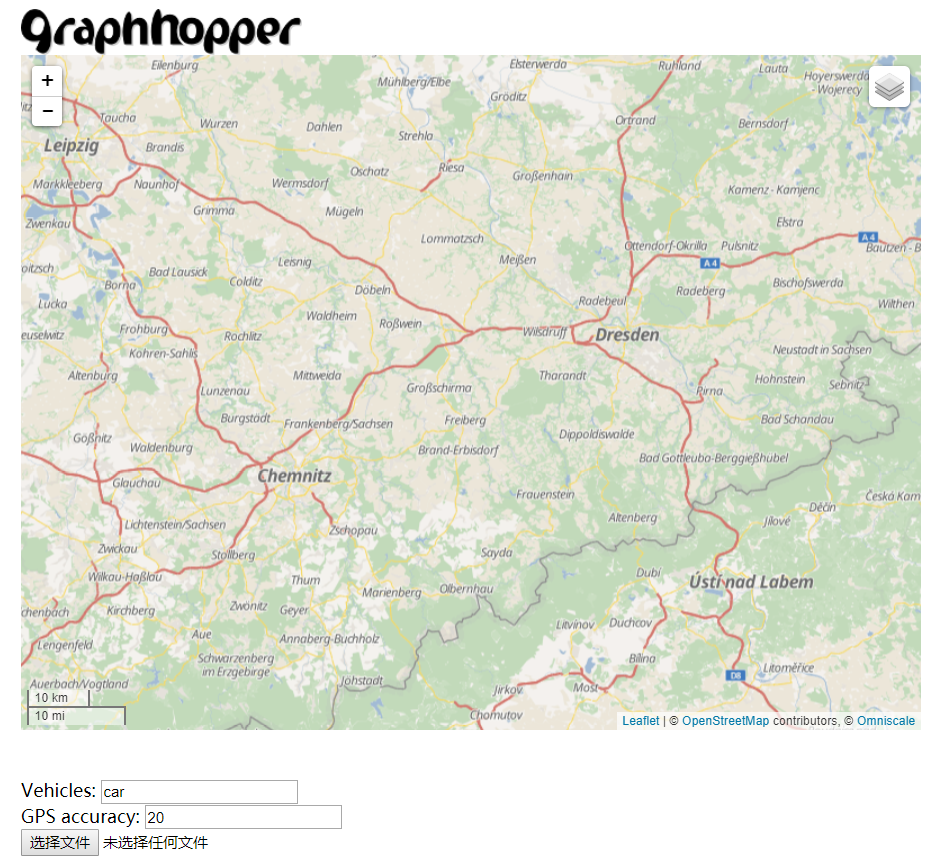
->GHUtil.js

->togeojson.js

->leaflet.js

### /index.html

运行截图：



用户输入数据：

1. **<input** type="text" value="car" id="vehicle-input"**/>**
2. **<input** type="number" value="20" id="accuracy-input"**/>**
3. **<input** type="file" id="matching-file-input"**/>**

### /Demo.js

创建地图L.map，即setup里面的map

1. function createMap(divId)
2. //leaflet.js <https://www.cnblogs.com/shitao/tag/leaflet/>

创建响应，发送到服务器

1. **var** mmClient = **new** GraphHopperMapMatching(/\*{host: "https://graphhopper.com/api/1/", key: ""}\*/);

处理输入数据：

1. **var** vehicle = $("#vehicle-input").val();
2. **if** (!vehicle)
3. vehicle = "car";
4. **var** gpsAccuracy = $("#accuracy-input").val();
5. **if** (!gpsAccuracy)
6. gpsAccuracy = 20;

读取gpx文件

1. document.getElementById('matching-file-input').addEventListener('change', readSingleFile, **false**);

解析gpx的xml文档为geo json格式，显示黑色道路数据（真实的gpx轨迹）

1. **var** dom = (**new** DOMParser()).parseFromString(content, 'text/xml');//解析xml文档
2. **var** pathOriginal = toGeoJSON.gpx(dom);//将gpx转换为GeoJSON
3. // GeoJSON是一种用于编码各种地理数据结构的格式，详见https://www.npmjs.com/package/togeojson
4. //https://www.npmjs.com/package/osmtogeojson
5. //togeojson.js
7. //http: geojson.org/geojson-spec.html
8. **if** (pathOriginal.features[0]) {
9. pathOriginal.features[0].properties = {style: {color: "black", weight: 2, opacity: 0.9}};
10. //真实的gpx轨迹为黑色线表示
11. routeLayer.addData(pathOriginal);//新增标记
12. $("#map-matching-response").text("calculate route match ...");//匹配后的返回信息
13. $("#map-matching-error").text("");
14. } **else** {
15. $("#map-matching-error").text("Cannot display original gpx file. No trk/trkseg/trkpt elements found?");
16. }

服务器地址

1. GraphHopperMapMatching = **function** (args) {//mmClient
2. **this**.host = "/";
3. **this**.basePath = "match";  //MapMatchingResource.java
4. **this**.vehicle = "car";
5. **this**.gps\_accuracy = 20;
6. **this**.data\_type = "json";
7. **this**.max\_visited\_nodes = 3000;
8. graphhopper.util.copyProperties(args, **this**);
9. };
10. **var** url = args.host + args.basePath + "?vehicle=" + args.vehicle
11. + "&gps\_accuracy=" + args.gps\_accuracy
12. + "&type=" + args.data\_type
13. + "&max\_visited\_nodes=" + args.max\_visited\_nodes;
15. **if** (args.key)
16. url += "&key=" + args.key;
18. $.ajax({
19. timeout: 20000,//设置请求超时时间
20. url: url,//发送请求的地址。
21. contentType: "application/xml",//发送信息至服务器时内容编码类型
22. type: "POST",//请求方式
23. data: content//发送到服务器的数据
24. }) ……………………

发送到服务器

1. mmClient.doRequest(content,
2. **function** (json) {...}, {gps\_accuracy: gpsAccuracy});

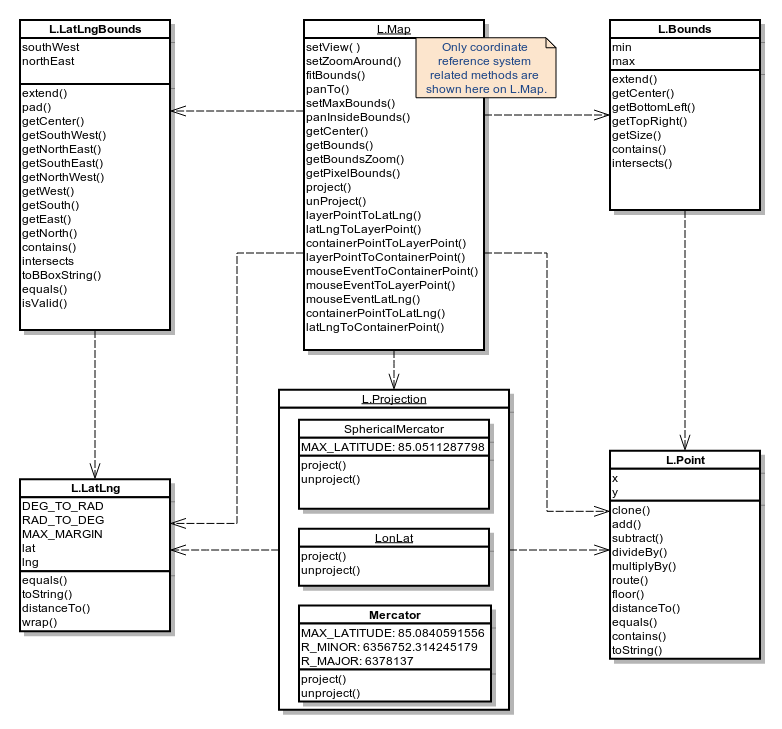
其中的function ( json ）即为匹配

显示绿色匹配道路数据，通过json从后台处理相关数据（$ajax）

1. **function** (json) {//关于json https://www.jb51.net/article/134072.htm// https://github.com/graphhopper/graphhopper/blob/master/docs/web/api-doc.md
2. **if** (json.message) {
3. $("#map-matching-response").text("");
4. $("#map-matching-error").text(json.message);
5. } **else** **if** (json.paths && json.paths.length > 0) {
6. **var** mm = json.map\_matching;
7. **var** error = (100 \* Math.abs(1 - mm.distance / mm.original\_distance));
8. error = Math.floor(error \* 100) / 100.0;
9. $("#map-matching-response").text("success with " + error + "% difference, "
10. + "distance " + Math.floor(mm.distance) + " vs. original distance " + Math.floor(mm.original\_distance));
11. **var** matchedPath = json.paths[0];
12. **var** geojsonFeature = {
13. type: "Feature",
14. geometry: matchedPath.points,
15. properties: {style: {color: "#00cc33", weight: 6, opacity: 0.4}}
16. };
17. //用绿色表示匹配的结果
19. routeLayer.addData(geojsonFeature);
21. **if** (matchedPath.bbox) {
22. **var** minLon = matchedPath.bbox[0];
23. **var** minLat = matchedPath.bbox[1];
24. **var** maxLon = matchedPath.bbox[2];
25. **var** maxLat = matchedPath.bbox[3];
26. **var** tmpB = **new** L.LatLngBounds(**new** L.LatLng(minLat, minLon), **new** L.LatLng(maxLat, maxLon));
27. map.fitBounds(tmpB);
28. }
29. } **else** {
30. $("#map-matching-error").text("unknown error");
31. }
32. }

### /leaflet.js

<https://www.cnblogs.com/shitao/tag/leaflet/>



## Matching-web-bundle

### /MapMatchingResource.java

#### 在远程客户端应用程序中使用GraphHopper映射匹配的资源。

1. @javax.ws.rs.Path("match")
2. **public** Response doGet(
3. Document body,
4. @Context HttpServletRequest request,
5. @QueryParam(WAY\_POINT\_MAX\_DISTANCE) @DefaultValue("1") **double** minPathPrecision,
6. @QueryParam("type") @DefaultValue("json") String outType,
7. @QueryParam(INSTRUCTIONS) @DefaultValue("true") **boolean** instructions,
8. @QueryParam(CALC\_POINTS) @DefaultValue("true") **boolean** calcPoints,
9. @QueryParam("elevation") @DefaultValue("false") **boolean** enableElevation,
10. @QueryParam("points\_encoded") @DefaultValue("true") **boolean** pointsEncoded,
11. @QueryParam("vehicle") @DefaultValue("car") String vehicleStr,
12. @QueryParam("locale") @DefaultValue("en") String localeStr,
13. @QueryParam(Parameters.DETAILS.PATH\_DETAILS) List<String> pathDetails,
14. @QueryParam("gpx.route") @DefaultValue("true") **boolean** withRoute /\* default to false for the route part in next API version, see #437 \*/,
15. @QueryParam("gpx.track") @DefaultValue("true") **boolean** withTrack,
16. @QueryParam("gpx.waypoints") @DefaultValue("false") **boolean** withWayPoints,
17. @QueryParam("gpx.trackname") @DefaultValue("GraphHopper Track") String trackName,
18. @QueryParam("gpx.millis") String timeString,
19. @QueryParam("traversal\_keys") @DefaultValue("false") **boolean** enableTraversalKeys,
20. @QueryParam(MAX\_VISITED\_NODES) @DefaultValue("3000") **int** maxVisitedNodes,
21. @QueryParam("gps\_accuracy") @DefaultValue("20") **double** gpsAccuracy)

#### 新建算法MapMatching 类：利用doWork函数进行运算

1. MapMatching matching = **new** MapMatching(graphHopper, opts);
2. matching.setMeasurementErrorSigma(gpsAccuracy);//赋值标准差
3. MatchResult matchResult = matching.doWork(gpxFile.getEntries());

#### Response返回值(matchResult) : "extended\_json".equals(outType)

1. Response.ok(convertToTree(matchResult, enableElevation, pointsEncoded)).
2. header("X-GH-Took", "" + Math.round(took \* 1000)).
3. build();

#### Response返回值(rsp) :

writeGPX==true

writeGPX="gpx".equalsIgnoreCase(outType);

1. pathMerger.doWork(pathWrapper, Collections.singletonList(matchResult.getMergedPath()), tr);
3. // clear empty paths.
4. pathWrapper.getErrors().clear();
5. GHResponse rsp = **new** GHResponse();
6. rsp.add(pathWrapper);
7. Response.ok(rsp.getBest().getInstructions().createGPX(trackName, time, enableElevation, withRoute, withTrack, withWayPoints, Constants.VERSION), "application/gpx+xml").header("Content-Disposition", "attachment;filename=" + "GraphHopper.gpx").header("X-GH-Took", "" + Math.round(took \* 1000)).build();

#### Response返回值(map) :

writeGPX==false

writeGPX="gpx".equalsIgnoreCase(outType);

1. Map<String, Object> matchStatistics = **new** HashMap<>();
2. matchStatistics.put("distance", matchResult.getMatchLength());
3. matchStatistics.put("time", matchResult.getMatchMillis());
4. matchStatistics.put("original\_distance", matchResult.getGpxEntriesLength());
5. matchStatistics.put("original\_time", matchResult.getGpxEntriesMillis());
6. map.putPOJO("map\_matching", matchStatistics);
7. Response.ok(map).
8. header("X-GH-Took", "" + Math.round(took \* 1000)).
9. build();

## Matching-core

### MapMatching.java的doWork函数

#### lookupGPXEntries函数

1. //过滤条目：
2. //滤掉过于相近的两个点，距离大于两个标准差
3. List <GPXEntry> filteredGPXEntries = filterGPXEntries（gpxList）;
5. //现在找到图表中的每个条目：
6. // A \*算法
7. //http://www.360doc.com/content/16/1201/12/99071\_610999046.shtml
8. List <Collection <QueryResult >> queriesPerEntry = lookupGPXEntries（filteredGPXEntries，DefaultEdgeFilter.allEdges（weighting.getFlagEncoder（）））;

对于每个点调用函数

1. findNClosest (gpxEntry.lat, gpxEntry.lon, edgeFilter, measurementErrorSigma);// 返回在GPS信号精度范围内所有非常接近的结果。

而后，对每个点两次调用函数（iteration=0/1）

1. findNetworkEntries(queryLat, queryLon, set, iteration);

遍历该点周围的九个格子

1. findNetworkEntriesSingleRegion(foundEntries, subqueryLat, subqueryLon);

iteration=0时，为第一个遍历的，iteration=1，依次遍历后八个，具体顺序如下表所示：

|  |  |  |
| --- | --- | --- |
| 2 | 8 | 3 |
| 4 | 1 | 5 |
| 6 | 9 | 7 |

对每个点：

1. **long** keyPart = createReverseKey(queryLat, queryLon);
2. //对经纬度编码（二叉树）后，将得到的编码反转（更容易算术右移）
3. fillIDs(keyPart, START\_POINTER, storedNetworkEntryIds, 0);
4. // START\_POINTER=1

set.add(……);

#### 初始化queryGraph

1. //向图表添加虚拟节点和边缘，以便可以表示边缘上的由虚拟节点产生的候选项。
2. QueryGraph queryGraph =  **new** QueryGraph（routingGraph）.setUseEdgeExplorerCache（**true**）;
3. List <QueryResult> allQueryResults =  **new** ArrayList <>（）;
4. **for** （Collection <QueryResult> qrs：queriesPerEntry）{
5. allQueryResults.addAll（QRS）; //添加所有找到的点
6. }

#### queryGraph.lookup函数

<https://github.com/graphhopper/graphhopper/blob/master/docs/core/low-level-api.md>

##### 什么是柱节点和塔节点？

对于像OSM这样的道路网络数据来说，方法几乎总是一条曲线，而不仅仅是一条直线。这些节点称为柱节点，位于塔节点(用于路由)之间，它们对于拥有更精确的几何形状是必要的。

那些交汇点节点（和死胡同的终端节点）我们称之为*塔节点*，它们也有一个相关的graphhopper节点ID，从0到graph.getNodes（）。结点之间的辅助节点我们称之为“支柱节点”

##### 什么是虚拟边缘和节点？

对于一条路线，不仅需要连接精度，即从塔节点到塔节点，但我们需要 GPS精确路线。为了使GPS精确路线成为可能，尽管我们从塔节点路由到塔节点，我们为位于边缘AB上的每个查询点引入一个新的虚拟节点x和虚拟边缘Ax，xB

\ /

A --- x --------- B

/ \

调用queryGraph.lookup(allQRs)将确定所有QueryResults 的正确节点：并创建新的虚拟节点，或者如果足够接近，则使用现有的节点。

具体执行步骤：

1. // 阶段1
2. //如果需要，计算最近边缘的捕捉点和交换方向
3. //阶段2 - 现在很清楚哪些点削减了一条边
4. // 1.创建点列表
5. // 2.在虚拟节点及其邻居（虚拟或普通节点）之间创建虚拟边缘

#### deduplicateQueryResultsByClosestNode函数

1. deduplicateQueryResultsByClosestNode(queriesPerEntry);//删除重复节点

#### createTimeSteps函数

1. // Creates candidates from the QueryResults of all GPX entries (a candidate is basically a
2. // QueryResult + direction).
3. //从所有GPX条目的查询结果中创建候选项(一个候选项基本上是QueryResult + direction)。
4. List<TimeStep<GPXExtension, GPXEntry, Path>> timeSteps =
5. createTimeSteps(filteredGPXEntries, queriesPerEntry, queryGraph);

为GPX条目创建候选时间步骤，但不创建发射或转移概率。为虚拟节点创建有向候选项，为实际节点创建无向候选项。

#### computeViterbiSequence函数 ——Viterbi\_Hmm

##### 计算最可能的匹配候选地图序列

1. List<SequenceState<GPXExtension, GPXEntry, Path>> seq = computeViterbiSequence(timeSteps, gpxList.size(), queryGraph);

##### 计算发射概率

1. **private** **void** computeEmissionProbabilities(TimeStep<GPXExtension, GPXEntry, Path> timeStep, HmmProbabilities probabilities) {
2. **for** (GPXExtension candidate : timeStep.candidates) {
3. // road distance difference in meters
4. **final** **double** distance = candidate.getQueryResult().getQueryDistance();
5. timeStep.addEmissionLogProbability(candidate,
6. probabilities.emissionLogProbability(distance));
7. }
8. }

##### 初始化概率

仅当HMM仅从第一次观察开始时才使用

1. viterbi.startWithInitialObservation(timeStep.observation, timeStep.candidates,timeStep.emissionLogProbabilities);

##### 计算转移概率

1. computeTransitionProbabilities(prevTimeStep, timeStep, probabilities, queryGraph);

##### 处理下一个时间步骤

（如果HMM broken，就不能调用它。）

1. viterbi.nextStep(timeStep.observation, timeStep.candidates,
2. timeStep.emissionLogProbabilities, timeStep.transitionLogProbabilities,
3. timeStep.roadPaths);

##### viterbi.isBroken()函数

判断Hmm 是否Broken，是则意味着所有状态的概率为零。

分析是否为两点之间距离过远

1. GPXEntry prevGPXE = prevTimeStep.observation;
2. GPXEntry gpxe = timeStep.observation;
3. **double** dist = distanceCalc.calcDist(prevGPXE.lat, prevGPXE.lon,
4. gpxe.lat, gpxe.lon);

##### 未broken则继续循环，遍历每个点

1. timeStepCounter++;//继续循环
2. prevTimeStep = timeStep;

##### 遍历每个点后的返回值

1. **return** viterbi.computeMostLikelySequence();

#### Matched real edges

1. **final** EdgeExplorer explorer = queryGraph.createEdgeExplorer(DefaultEdgeFilter.allEdges(weighting.getFlagEncoder()));
2. **final** Map<String, EdgeIteratorState> virtualEdgesMap = createVirtualEdgesMap(queriesPerEntry, explorer);
3. MatchResult matchResult = computeMatchResult(seq, virtualEdgesMap, gpxList, queryGraph);

##### createEdgeExplorer函数——graph.java接口中的函数

返回一个EdgeExplorer，它使遍历特定节点的所有过滤边缘成为可能。尽可能减少对这个方法的调用，例如在for循环之前创建一个explorer。

graph.java接口：表示(geo)图的接口——适合于高效存储，因为它可以通过称为节点id的索引请求。要获得lat、lon点，您需要建立一个LocationIndex实例。

函数中调用函数createUncachedEdgeExplorer(edgeFilter);

##### createVirtualEdgesMap函数

返回一个映射，其中每个虚拟边缘以正确的方向映射到其实际边缘。

##### computeMatchResult函数

返回匹配结果集

### MapMatchingTest.java的testDoWork()

第一步：创建mapmatching类、GPX点集，调用doWork函数

1. MapMatching mapMatching = **new** MapMatching(hopper, algoOptions);//创建类mapmatching
2. List<GPXEntry> inputGPXEntries = createRandomGPXEntries(//创建两个测试gpx点
3. **new** GHPoint(51.358735, 12.360574),
4. **new** GHPoint(51.358594, 12.360032));
5. MatchResult mr = mapMatching.doWork(inputGPXEntries);//调用dowork函数

第二步：确保没有虚拟边缘

1. // make sure no virtual edges are returned
2. **int** edgeCount = hopper.getGraphHopperStorage().getAllEdges().length();//所有边的数量
3. **for** (EdgeMatch em : mr.getEdgeMatches()) {//循环每条已匹配出的边
4. assertTrue("result contains virtual edges:" + em.getEdgeState().toString(), em.getEdgeState().getEdge() < edgeCount);  //当前边的边id < 所有边的总数
5. }

第三步：创建道路名字，比较误差

1. // create street names
2. assertEquals(Arrays.asList("Platnerstraße", "Platnerstraße", "Platnerstraße"), fetchStreets(mr.getEdgeMatches()));
3. //原始gpx道路长度与地图匹配的道路长度，误差1.5之内，以米为单位
4. assertEquals(mr.getGpxEntriesLength(), mr.getMatchLength(), 1.5);
5. //原始GPX轨道的长度与地图匹配的道路长度，以毫秒为单位
6. assertEquals(mr.getGpxEntriesMillis(), mr.getMatchMillis());

第四步：PathMerger.doWork为matchGHRsp添加mr的path，并简化路径

1. PathWrapper matchGHRsp = **new** PathWrapper();
2. //简化路径： PathSimplification
3. **new** PathMerger().doWork(matchGHRsp, Collections.singletonList(mr.getMergedPath()), SINGLETON.get("en"));
4. InstructionList il = matchGHRsp.getInstructions();

  第五步：比较InstructionList 的il

1. assertEquals(il.toString(), 2, il.size());
2. assertEquals("Platnerstraße", il.get(0).getName());