マップマッチングの説明

ここでは，取得したプローブデータからマップマッチングを行うための Java のコードの内容と行っている計算について説明します．マップマッチングとは，図-1 のように GPS などで観測した位置データの誤差を補正して，道路ネットワーク上で移動経路を特定することを指します．

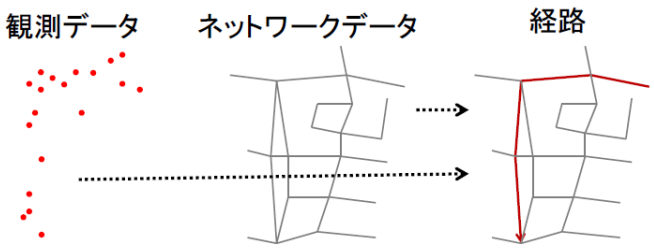

図 1 マップマッチングのイメージ図

今回紹介するマップマッチングのプログラムの概要は以下のようになっています．

入力：ネットワークデータ（ノード・リンク）・PP データ（トリップデータ・ロケーションデータ）
①トリップ単位でロケーションデータを分割する
②ネットワーク内の対象リンクのリンク尤度を計算する
③ネットワーク内の対象リンクの中から，起点リンクと終点リンクを決定する
④リンク尤度をリンクコストとして最短経路探索をすることで，経路を決定する
出力：経路データ・リンクデータ・kml 形式のデータ

1 プロジェクトとデータの準備

まず，プロジェクトとデータの準備をします．「Mapmatching」のファイルを任意の場所に保存します（workspace の下など）．次に，ダウンロードした「input_mapmatching」の「input」のファイルの中に入っているデータを「Mapmatching」の「input」のファイルに保存します(/～/Mapmatching/input/～)．

ここまでの準備が出来たら，eclipse を起動し，プロジェクトのインポートをして(「Java の導入」に記載してあるのでそれを見ていただけるとわかりやすいと思います)，Main.java ファイルを開きます．Main.java ファイルと実行することでマップマッチングのプログラムを動かすことが出来ます．

以下では，Main.java ファイルのコードに従って解説を進めていきます．

1 データファイルの読み込み

```
18  //読み込みファイルの指定
19  String file1 = "./input/YOKOHAMAnode.csv";     //nodeデータの場所を指定
20  String file2 = "./input/YOKOHAMAlink.csv";     //linkデータの場所を指定
21  String file3 = "./input/UnlinkedTrip.csv";     //tripデータの場所を指定
22  String file4 = "./input/LocData.csv";          //location データの場所を指定
```

まず，入力データファイルの設定を行います．入力データファイルを格納した場所とファイル名を上図のように設定します．データの場所は，ファイル上で右クリック→プロパティから確認することができます．ディレクトリ(フォルダ)を表す記号が”¥”ではなく，”/”であることに注意してください．データを読み込む際には，「,」で区切ってあるデータを一列ずつ項目ごとに読み込みます．このデータの順序が変更されるとデータの読み込みに失敗してしまいます．以下に入力データについて説明があるので，自分でサンプルデータを作成する際には，以下を参考にして作成してください．

2 入力データについて

入力データは，ノードデータ，リンクデータ，トリップデータ，ロケーションデータの4つのデータを使用します．ここで，ノードデータ，リンクデータに関して，道路網はノードとリンクによって表現され，ノードとリンクには固有の ID 番号が振られています．リンクは方向毎に起終点が分けられており，一本の道路に対して，双方向に通行可能な場合は，上下線の2本のリンクに分かれています．（ノード1・ノード2間の道路の場合，1→2と2→1のふたつのリンクが存在します）ここでは，計算時にリンクの O と D を区別しているため，リンクが一方通行かどうかを考慮する必要はありません．

◇　ノードデータ（YOKOHAMAnode.csv）
1 列目：ノード ID(nodeID)…各ノードを識別する ID
2列目：各ノードの緯度(lat)…北緯，日本測地系
3列目：各ノードの経度(lng)…東経，日本測地系

◇　リンクデータ（YOKOHAMAlink.csv）
1 列目:リンク ID(LinkID)
2 列目:出発ノード(O)
3 列目:到着ノード(D)
4 列目:車線数(lanes)

◇トリップデータ（UnlinkedTrip.csv）
1 列目：アンリンクドトリップ ID…最初のアンリンクドトリップはトリップ ID +「011」　・前のアンリンクドトリップと交通手段が一緒なら 1 の位を増やす．
（例:「011」→「012」）

・前のアンリンクドトリップと交通手段が異なるなら 10 の位を増やして，1 の位を 1 にする．
（例:「012」→「021」）

2列目：モニターID…トリップを行ったモニターの ID

3列目：TripID…各トリップに割り振られている ID

4列目：移動開始時刻

5列目：移動終了時刻

6列目：交通手段…自動車での移動のコード番号は以下の通りです．

000	--	600	食事
100	出勤	700	娯楽
200	帰宅	800	散歩・回遊
300	帰社	900	サイクリング
400	業務	910	モビリティカフェ
500	買い物	999	その他

◇ロケーションデータ(locData.csv)

1列目：測位データ ID…各測位データの ID

2列目：ユーザーコード…ユーザーに割り振られているコード番号

3列目：測位日時

4列目：緯度（日本測地系）

5列目：経度（日本測地系）

6列目：測位モード…データの取得方法を表しています．（0：GPS，1：Hybrid，2：複数基地局，3：測位失敗）

3 出力ファイルの設定

```
40  //マップマッチング計算
41  try {
42       PrintWriter line1 = new PrintWriter(new FileWriter("./output/link.csv"));
       //全体の経路データ
43       line1.println("モニターID,ダイアリーID,交通手段,リンクID,リンク出発時刻,所要時間(s),速
   度(km/h),リンクコスト,リンク長(m)");
44       PrintWriter line2 = new PrintWriter(new FileWriter("./output/path.csv"));
       //全体の経路データ
45       line2.println("モニターID,ダイアリーID,交通手段,経路出発nodeID,経路到着nodeID,経路長
   (m),所要時間(s),速度(km/h)");
46       List <Trip> tripList = ReadFile.getTripList();
47      for(Trip trip : tripList){
48          //モニターごとにフォルダーを作成しその中に管理
49          File file00 = new File("./output/linkFile/" + trip.getPersonalID() + "/");
50          //もしそのモニター名のフォルダがないなら作成する
51       if(file00.exists() == false && trip.getLocationList().size() > 1){
52          file00.mkdirs();
53       }
```

ここでは，出力ファイルの指定をします．あらかじめ「Mapmatching」の直下のディレクトリに「output」ファイルを作成しておきます．コンパイルすると，ここに「link.csv」「path.csv」「linkFile」という名前の出力データが保存されます．算出されたデータは，43，45 行目のカンマで区切られた各項目に格納されます．

4 出力データについて

出力データは「link.csv」「path.csv」「linkFile」の3種類です．もともとのインプットデータには，自動車以外での移動によるトリップも含まれていますが，使用しているネットワークは車道ネットワークなので，プログラムでは自動車トリップのみを読み込んでいます．

◇リンクデータ(link.csv)

- モニターID…トリップを行ったモニターの ID
- ダイアリーID…各トリップの ID
- 交通手段…各交通手段のコード番号
- リンク ID…各リンクに割り振られている番号
- リンク出発時刻
- 所要時間(s)
- 速度(km/h)
- リンクコスト…算出されたリンクコスト
- リンク長(m)

◇パスデータ(path.csv)

- モニターID…トリップを行ったモニターの ID
- ダイアリーID…各トリップの ID
- 交通手段…各トリップの ID
- 経路出発 nodeID…トリップを開始したノードの ID
- 経路到着 nodeID…トリップを終了したノードの ID
- 経路長（m）
- 所要時間(s)
- 速度(km/h)

◇linkFile

モニター毎にフォルダが作成されてあり，kml 形式でデータが格納されています．KML (Keyhole Markup Language) は、GoogleEarth や Google マップなどのアプリケーションに表示するポイント、線、画像、ポリゴン、モデルなどの地理的特徴をモデリングして保存するためのファイル形式です。KML ファイルは GoogleEarth にドラッグ＆ドロップすることで，ユーザーの移動経路を表示させることが出来ます．

5 リンク尤度の算出

読み込んだノード･リンクデータと抽出したトリップデータから，経路の精度を評価するために，リンク尤度を算出します．図-2 にリンク尤度の算出のイメージ図を示します． リンク尤度の計算は Matching.java ファイルの中で行われており，式(1)によって求めます．

$$LL = (l_{upN} + L_{dnN} + \alpha \times l_{mid}) \times \frac{L_{length}}{\beta \times L_{lane}} \quad \cdots(1)$$

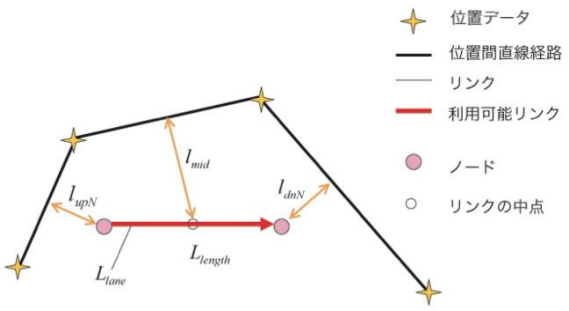

図 2 リンク尤度算出のイメージ図

ここで，

LL :リンク尤度

l_{upN} :起点ノードから位置間直線距離までの最短経路

l_{dnN} :終点ノードから位置間直線距離までの最短経路

l_{mid} :リンク中央から位置間直線距離までの最短経路

L_{length} :リンク長

L_{lane} :リンク車線数

```
55  if(trip.getLocationList().size()> 1){
56      //リンクの中点から取得点を繋げたリンクまでの最短距離を算出し，起終点および中点と取得点を
          つなげた経路の面積をリンクコストとして算出
57     //初期値
58     int minCost = Integer.MAX_VALUE;
59     int limit = DISTANCE_TO_NODE; //ノードとの距離
60     while(true){
61           //ネットワーク内の対象リンクを抽出しコストを算出する
62           Matching.exeLinkCost(ReadFile.getNodeMap(), ReadFile.getLinkMap(),
    trip.getLocationList(), limit);
63
64           //ネットワーク内の対象リンクの中から起点リンク候補を決定する
65           List<Integer> startLinkList = null;
66           for(int j = 0; j < trip.getLocationList().size();j++){
67                if(startLinkList==null || startLinkList.isEmpty()==true){
68                     startLinkList = Matching.getNearLinkList(ReadFile.getLinkMap(),
    trip.getLocationList(), j);
69                }
70           }
71
72           //ネットワーク内の対象リンクの中から終点リンク候補を決定する
73           List<Integer> endLinkList=null;
74           for(int j = trip.getLocationList().size() ;j > 0; j--){
75                if(endLinkList==null || endLinkList.isEmpty()==true){
76                     endLinkList=Matching.getNearLinkList(ReadFile.getLinkMap(),
    trip.getLocationList(), j-1);
77                }
78           }
```

6 最短経路探索による経路の特定

どの経路を通ったのかを特定する計算方法について説明します．まず，位置間直線経路(ロケーションデータを直線で結んだもの)の起点および終点から最も近いノードをそれぞれ起点ノード(O)，終点ノード(D)とします．そして，求めたリンク尤度をリンクコストとして最短経路探索を行います．最短経路探索はダイクストラ法と呼ばれるアルゴリズムを利用し，計算は，Dijkstra.java ファイルの中で行われています．図-3 に最短経路探索による経路特定のイメージ図を示します．

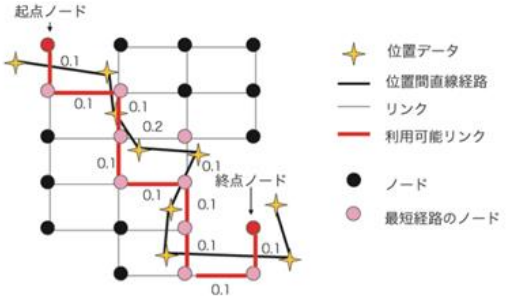

図 3　最短経路探索法による経路の特定

```
 //起点，終点のリンク候補同士から経路列挙
for(int j = 0; j < startLinkList.size();j++){
    int sLinkID = startLinkList.get(j);
    int sNodeID = ReadFile.getLinkMap().get(sLinkID).getUpNodeID();
    for(int k = 0; k < endLinkList.size(); k++){
        int eLinkID = endLinkList.get(k);
        int eNodeID = ReadFile.getLinkMap().get(eLinkID).getDnNodeID();
        if(sNodeID != eNodeID){
            System.out.println(sLinkID+" "+eLinkID+" "+sNodeID+" "+eNodeID);
            Dijkstra.exeDijkstra(ReadFile.getLinkMap(),
ReadFile.getNodeMap(), sNodeID,eNodeID);
            int cost = Dijkstra.getCost(ReadFile.getNodeMap(), eNodeID);
            if(minCost > cost){
               minCost = cost;
               trip.setStartNodeID(sNodeID);
               trip.setEndNodeID(eNodeID);
            }
        }
     }
}

```

7 データクリーニング

最後に，マッチング後にデータの分析を進めて行く上で，不必要なトリップデータが生成されることがあるため，データクリーニングを行います．データクリーニングの必要があるデータとして，出発ノードと到着ノードが同じもの，出発ノードと到着ノードが同一のもの，出発地と到着地がつながっていないもの，自動車トリップとしての取得点がないものとしました．最終的にはマッチングできたものだけ結果を出力します．