Public-Transit Data Extraction from OpenStreetMap Data

Zhiwei Zhang

09.1.2017
Bearbeitungszeitraum

Gutachter
Prof. Dr. Hannah Bast

Betreuer
Patrick Brosi
Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.
## Contents

Abstract 1  
Zusammenfassung 2  
Acknowledgements 3  

1 Introduction 4  
  1.1 Motivation 4  
  1.2 Brief Procedure 4  

2 Related Work 8  

3 Algorithms and Data Structures 10  
  3.1 Data Structures 10  
    3.1.1 OSM Data 10  
    3.1.2 Network 10  
    3.1.3 GTFS 12  
    3.1.4 GTFS System 12  
    3.1.5 R-Tree 13  
  3.2 Algorithms 14  
    3.2.1 Dijkstra’s Algorithm 14  
    3.2.2 A* Search 14  
    3.2.3 Bidirectional A* Search 15  
    3.2.4 Depth-First Search (DFS) 15  
    3.2.5 Support Vector Machines (SVMs) 15  

4 Public Transit Data Extraction 16  
  4.1 Data Extraction 16  
  4.2 Repair System 16  
    4.2.1 Order Repair 18  
    4.2.2 Gap Repair 19  
    4.2.3 Classification 19  
    4.2.4 Connection Repair 21  
    4.2.5 Topological Sort 21  
  4.3 Generating GTFS Feed 22  

5 Evaluation 24
6 Further Research

6.1 Extending network by collecting all the ways near to at least one of the railways ........................................ 28
6.2 Improving the efficiency of classification .................................................. 28
6.3 Identifying neighbors with regard to the existence of a real gap ........ 28

Bibliography ........................................... 30
Abstract

The major objective of this paper is to acquaint the reader with details on how to extract public transit data from OpenStreetMap\(^1\) (OSM) data. To achieve this aim, we implement a program extracting train-related data from an OSM file and introduce its development to help the reader understand our method.

Its development is broken into three basic incremental steps. In the first step, we search for the information related to trains with the assist of some particular attributes and store it into a railway network. In the second step we fix attention on completing the network step by step through solving some problems, i.e., incorrect order, gaps, missing class label and so on. In the last step we transform the network into a General Transit Feed Specification\(^2\) (GTFS) feed. Through uploading the feed to Transit Visualization Client\(^3\) (TRAVIC) and OpenTripPlanner\(^4\) (OTP), this network turns visible and we can run some routing test. The visible displayed result in TRAVIC and test result in OTP demonstrate that our method works as expected. Due to the limitation of OSM data, some gaps are irreparable and bias exists in distinguishing between regional railway and long-distance railway. Therefore, further research is needed to improve the performance of this method.

\(^1\)OpenStreetMap is an open source object providing the map data.
\(^2\)General Transit Feed Specification defines a common format for public transportation schedules and associated geographic information.
\(^3\)Transit Visualization Client provides movement visualization of transit data published by transit agencies and operators from all over the world.
\(^4\)OpenTripPlanner is an open source platform for route planning.
Zusammenfassung

Das Hauptziel dieser Arbeit ist es, den Leser mit Details über die Extrahierung öffentlicher Transitdaten aus OpenStreetMap (OSM) Daten vertraut zu machen. Um dieses Ziel zu erreichen, implementieren wir ein Programm, das zug-bezogene Daten aus einer OSM-Datei extrahiert und wir führen seine Entwicklung ein, um dem Leser zu helfen, unsere Methode zu verstehen.

Acknowledgements

I would like to thank my supervisor Prof. Dr. Hannah Bast for giving me the opportunity to choose this interesting topic, for enlightening me in programming and for showing me the enchantment of computer science. Furthermore, I want to thank my tutor Patrick Brosi for proposing useful ideas, for answering my questions and for his generous help. Finally, I want to thank my parents and my wife for supporting me.
1 Introduction

1.1 Motivation

As an open-source project OSM data has been enhanced considerably by worldwide contributors in recent years. Also, its openness makes it available for many applications to supply various services to people, i.e., navigation ([HMH07]; [Vet10]), route planing ([BCE10]; [BDPW13]), real-time traffic state evaluation [TMRR12], catastrophe caution [RAC12] and so on. However, not entire but partial data such as public transit part can already satisfy some of those applications’ demands. Taking this situation into account, extracting public transit data from OSM data is necessary.

Currently there are dozens of GTFS feeds for various regions all over the world. But none of them takes aim at long-distance public transport (mostly trains). Thus, generating a GTFS feed for long-distance public transit can not only fill a void but also enable developers to test some new routing algorithms on it. Especially, we want to give support for testing the new big route planing project, which is developed by Dr. Prof. Bast’s research group.

For two aforementioned reasons, this thesis focuses mainly on extraction of long-distance public transport data and generation of a GTFS feed for Europe’s transit network.

1.2 Brief Procedure

We give a brief overview of our method to make the reader have a better understanding.

First of all, we process the OSM file to separate necessary information, then use it to build our public transit network.

After construction, we use a designed repair system to remove existing problems in the network. The repair system includes five parts: Order Repair, Gap Repair, Classification, Connection Repair, Topological Sort.

1. Order Repair aims mainly at solving the incorrect order problem through bidirectional A* algorithm, i.e. move as many as possible partial paths in position inside a railway.

2. In Gap Repair part we fill gaps with some partial paths by using path finding algorithms.
Figure 1.1: A rendered picture of railways in OpenStreetMap.
3. Classification takes charge of assigning each railway to an appropriate class: local or long-distance.

4. Connection Repair is responsible for normalizing connections between partial paths in each railway.

5. In Topological Sort part all the partial paths in each railway are stored as a topologically sorted list.

More details about the repair system and each part of it will be explained in chapter 4.2.

Eventually, we transform the public transit network into a GTFS feed.

\[\text{Data Extraction} \rightarrow \text{OSM Data} \]

\[\text{Repair System} \rightarrow \text{Order Repair} \rightarrow \text{Gap Repair} \rightarrow \text{Classification} \rightarrow \text{Connection Repair} \rightarrow \text{Topological Sort} \rightarrow \text{Public Transit Data} \rightarrow \text{Generating GTFS Feed} \rightarrow \text{A GTFS Feed} \]

**Figure 1.2:** A picture of the brief Procedure.

As a part of this thesis the program *ExtractionFromOSM* is provided. It’s able to extract train-related data from an OSM file using the above mentioned method, will be introduced as a concrete example for the explanation of this method in the
1.2 Brief Procedure

following text. This program consists of about 6000 lines of code in C++. Functional functions are mostly tested with Google Test.

Figure 1.3: A picture of TRAVIC.
Those colorful points are public transport vehicles in New York.
2 Related Work

Five papers are relevant to the topic of this thesis. For each of them a brief introduction is given in following.

In 2010 Bast et al. [BCE+10] described a new method for routing on public transportation networks. The basic idea is that all optimal paths are found out first, then we cut them into parts and store these parts. So when an optimal path from a given source station to a given target station is queried, all necessary parts are combined to build a partial network containing this optimal path. Then finding the optimal connection(s) amounts to a shortest-path computation on the network. Owing to this precomputation of all optimal paths, this method is fast even when the network is realistically modeled, very large and poorly structured.

Since OSM is the biggest open-source map project and provides free map data including the public transport data, it’s the perfect candidate providing data-support for testing or running the above mentioned fast routing method at present.

However, for lack of communication between OSM community and transport agencies, the quality of public transit data in OSM data was not optimal several years ago. To change this situation in 2011 Tran et al. [THBL11] designed a new framework named GTFS-OSM Synchronization (GOSync). With its help the GTFS datasets of transport agencies become open and can be enhanced by open-source communities such as the OSM community. The experimental results provide a strong proof that this framework promotes the development in the open sharing and improvement of public transportation information.

In recent years, OSM data has developed rapidly, but it still contains some errors. Funke et al. [FSS15] developed a new devised classifier in 2015, which is used for detecting gaps in a road. Also, they illustrated the details on using machine learning techniques to find road segments where the name tag can be extrapolated with high probability. Their experimental results evidence that their methods contribute significantly to the enhancement of the quality of OSM road network data.

And Sehra et al. [SSR16] published a paper in 2016, which mainly focuses on detecting topological errors in OSM data. They classified the common topological errors according to the type of features, i.e., point, linestring (line), polygon. And several algorithms are involved in the detection of these topological errors. The results prove the ability of their methods to identify the topological errors from Punjab map data successfully.

Although errors can be found, removing them is the real challenge and needs to be researched in the future.

Another related paper was written by Jorge Gil [Gil15] in 2015. In this paper
he portrayed the process of building a multi-modal urban network model using the OSM street network data set as its main structure. Onto this structure additional public transport and land use data sets are connected. He used the Randstad region data of the Netherlands as an example to exhibit the results of his work.
3 Algorithms and Data Structures

Before we start to illustrate the development of the program ExtractionFromOSM in detail, we need to put a concise interpretation of the involved algorithms and data structures.

3.1 Data Structures

3.1.1 OSM Data

OSM data is a particular XML file. It comprises of 5 XML-node types: root, bound, node, way and relation [OSM]. In the following text, we use root, bound, node, way and relation to denote the corresponding type XML-node respectively. Each OSM data has exactly one root and one bound, but there is no limit for other types.

- **Root** contains the version and encoding type information.
- **Bound** marks the geographical boundary of this file by limiting the minimum and maximum of latitude and longitude.
- A **node** means a geographical point on earth. It has a coordinate (latitude, longitude), an unique id and may have some attributes.
- A **way** includes a unique id and a list of node-ids. And it indicates a path which is produced by connecting the coordinates of the nodes in order of its node-id list.
- A **relation** corresponds to a route consisting of paths in reality. It contains an unique id, a list of way-ids and some attributes.

In standard OSM data all nodes precede all relations and all ways are in between them.

3.1.2 Network

Since we only need a subset of nodes, ways and relations in OSM data, we can store the information easily using an OSM-data-like data structure i.e. network. In network we use vertex, edge and graph to represent node, way and relation respectively. In this thesis a graph stands for a route taken by a train i.e. a railway, and we have two reasons for using the name “graph”:
3.1 Data Structures

1. it corresponds to the names “vertex” and “edge”.

2. “route” and “railway” are already used as class names in GTFS system.

In particular, each vertex, edge or graph can be sought out by its id in network, and equal vertexes are recorded in network as well. If two vertexes are close enough i.e. the distance between them is smaller than 1.5 meters, we treat them as identical ones.

The difference between graph and network is that graphs comprise network, and network unites all the graphs. In other words, graph is to network as relation is to OSM data.

![Diagram of network, graph, edge, vertex](image)

**Figure 3.1:** A picture of network, graph, edge, vertex.

Each color line indicates a graph (railway) in network.
3.1.3 GTFS

GTFS is the abbreviation of General Transit Feed Specification [GTF]. It defines a common format for public transportation schedules and associated geographic information. A GTFS feed contains 13 CSV files (with extension .txt) totally, but some of them are optional. To help the reader understand the “Generating GTFS Feeds” part in chapter 4.3, we give a brief introduction for the files relevant to this thesis.

- **agency.txt**: reserves the basic information of agencies in this feed, such as id, name, url, timezone, phone and language.
- **calendar.txt**: defines weekly service times. Each service time corresponds to a service_id.
- **routes.txt**: includes id, name, type, color and agency_id. It connects agencies with their trips.
- **shapes.txt**: stores many geographical points. Those geographical points represent geographical lines, each one of them defines a route taken by public transport.
- **stops.txt**: contains id, name, latitude and longitude of each stop.
- **stop_times.txt**: represents a stop sequence for each trip. Especially, an arrival_time and a departure_time are assigned to each stop.
- **trips.txt**: keep a list of stops for each route in reserve. Each trip references a service_id in calendar.txt.

3.1.4 GTFS System

GTFS system is a mapping between network and a valid GTFS feed. If we regard it as a factory, it takes a network as raw material and outputs a GTFS feed. It contains 5 classes:

- **railway**: takes over the information of a graph’s branch and waits for separating the information into other classes.
- **route**: corresponds to route.txt, and stores the basic attributes of a graph.
- **trip**: gathers information for trips.txt and stop_times.txt. It contains a sequence of stops, and each one of them has an arrival_time and a departure_time.
- **shape**: is homologous to shapes.txt, and saves a sequence of geographical points which defines a railway.
- **stop**: maps to stops.txt, and includes name, id, latitude and longitude of a station.
3.1 Data Structures

Figure 3.2: A picture of mapping relations among OSM data, network, GTFS system, a GTFS feed.

3.1.5 R-Tree

R-Tree [Gut84] is a tree data structure supporting many spatial access methods. In this thesis it is used mainly for indexing geographical coordinates, segments and rectangles. The basic idea of an R-Tree is to hold the geographical close feature of the objects. In an R-Tree the real objects are at the leaf level and other nodes are rectangles. A rectangle is a bounding box, which is the smallest one to keep all its members inside.

In case of inserting a new object two probabilities should be considered. The first one is the object to be inserted is totally inside the bounding box of another object, then it’s added to the corresponding node. The second one is an appropriate bounding box doesn’t exist, then a node is chosen to be extended by a heuristic function. If the chosen node beyonds its capacity of children nodes, it’s separated into two new nodes and replaced by them. This will be repeated from leaf to root until no node is out of its capacity.

The main idea about searching in R-Tree is to use the bounding boxes to decide whether or not to search inside a subtree. The approach of searching is just checking if the query intersects with the rectangles. If it is, then do it again with all children nodes of this intersecting one. Repeat this step down to the leaf level, then finding out all the intersecting objects.
3.2 Algorithms

3.2.1 Dijkstra’s Algorithm

Dijkstra’s algorithm [Dij59] is mostly used for finding the shortest path between nodes in graph.

**Procedure**:

1. The distance of the source node is set to 0 and the others to infinity. Then we set the statuses of all nodes to *unvisited* and create a *unvisited* set to keep them.

2. We get the node with minimal distance-value from the *unvisited* set and calculate the temporal distance for each *unvisited* neighbor of it. Here the distance of a node means the distance from current node to the source node. The calculation can be expressed by the formula:

   \[ \text{temporal\_dis}(v) = \text{dis}(u) + \text{distance}(u, v) \]  

   where \( v \) is the neighbor of the current node \( u \), \( \text{dis}(u) \) is the distance of current node \( u \), and \( \text{distance}(u, v) \) is the distance between current node \( u \) and its neighbor \( v \). After that we need to compare the temporal distance value \( \text{temporal\_dis}(v) \) with the current distance value \( \text{dis}(v) \). If \( \text{temporal\_dis}(v) < \text{dis}(v) \), we update the distance value of \( v \) with its temporal distance value and set \( u \) as the predecessor of \( v \). If the calculation and comparison for all the neighbors of \( u \) have been done, we set the status of \( u \) to *visited* and delete it from the *unvisited* set.

3. Repeat the second step until either the target node is *visited* or the minimum distance value in *unvisited* set is infinity.

3.2.2 A* Search

A* search [PEHR68] is a best-first search and widely used on weighted graph in path finding. It guarantees that its result-path always has the smallest cost.

**Procedure**:

1. Set the cost of the source node to 0 and insert it into the priority queue *open\_list*. A priority queue is a data structure in which the items with higher priority are always on the top and will be popped up first. The priority of a node is estimated by the addition of its cost and its heuristic. The formula expression is

   \[ f(x) = g(x) + h(x) \]  

   where \( x \) is the last node on the path, \( g(x) \) is the cost of the path from the start node to the current node, and \( h(x) \) is the heuristic, which evaluates the cost of the cheapest path from current node to target node.
2. The node with highest priority (i.e. lowest cost) is explored and deleted from \textit{open\_list}. The exploration of a node has three steps: estimating the priority for each one of its unexplored neighbors, inserting them into the \textit{open\_list} and marking the current node as their parent. If a neighbor is already in \textit{open\_list}, we compare its new cost with its current cost. If the new one is smaller than the old one, we update its cost and mark the current node as its parent.

3. The 2. step is repeated until target node is found or all reachable nodes from source node are explored.

### 3.2.3 Bidirectional A* Search

It’s a variant of A* search and its main procedure is similar to A*. The difference is bidirectional A* runs A* search simultaneously on source node and target node. Its stop condition is two paths are conjoint or all reachable nodes from both side are explored.

### 3.2.4 Depth-First Search (DFS)

DFS [THC01] is oriented to generate a topologically sorted node-list in a graph. Depth-first means that the children nodes of current node are always visited before other nodes at the same level of current node.

**Procedure**:

1. Insert the root node into a first-in-first-out (FIFO) queue and set all nodes as \textit{unvisited}.

2. Pop up top node from FIFO queue. If it’s \textit{unvisited}, insert its \textit{unvisited} neighbors into FIFO queue from behind and mark it as \textit{visited}.

3. Repeat step 2 until the FIFO queue is empty.

### 3.2.5 Support Vector Machines (SVMs)

SVMs [CV95] are widely used for two-class classification and regression. They consist of a learning model and a predicting model.

The basic idea of SVM is to find a hyper plane according to the training points and use it to predict which class every point to be predicted belongs to. The hyper plane divides the training points into two part, such that all the points in one part belong to exactly one class and the classes indicated by two parts are different. The distance from the nearest margin point of one part to this hyper plane is equal to another and its value must be maximum. In the predicting model we judge which side of the hyper plane the point falls on.
4 Public Transit Data Extraction

In this chapter we concentrate our attention on giving a detailed account of our method.

4.1 Data Extraction

Since only relation represents a train-route in OSM data, we begin data extraction by locating those relations which are related to trains. One observable common feature of the desired relations is that they contain exactly one of those attributes route=railway, route=rail, route=tracks, and route=train. Once we find one, store it in form of a graph in network. And for every node-id or way-id, which is included in this relation but not in network yet, a vertex or an edge is created and directly inserted into network. But for those relations with attribute route=tracks, we merely save it inside node-ids or way-ids through vertex or edge in network.

One thing to note is that in order to save time a tentative file “way.osm” is created with all the ways, while parsing the OSM file for the first time. Otherwise, we must pass all the nodes to reach ways in the original file.

In the next step, we track the ways from “way.osm” file, which satisfies one of two requirements:

1. its id has a corresponding edge in network
2. it has the attribute railway=rail or railway=narrow_gauge

When such a way is found, its information is used for supplementing the corresponding edge in network. And we also add a vertex to network for the vertex-id, which has been included by this way but not network.

Once more, we search for the nodes from original OSM file, whose id occurs in network. And with their information we complement the corresponding vertices in network.

Finally, we delete the file “way.osm”. So far we have finished the normal foundation work of the network.

4.2 Repair System

In this thesis, two edges are conjoint means the distance between them is smaller than 1.5 meters. If two edges are conjoint and their distance is bigger than 0
4.2 Repair System

meter, we must find the closest two end-vertexes of them and define these two end-vertexes as equal vertexes. Moreover, in reality a railway is continuous, which means any two index-adjacent edges in each graph are conjoint. If they are not, we found a gap. However, in network there are a lot of graphs containing gaps. The existence of a gap can be attributed to two actions:

1. one relevant edge is out of position.
2. some relevant edges are missing.

Here we regard the gap caused by the first action as a fake gap, the one caused by the second as a real gap. Beyond that, each railway could be classified as either local or long-distance according to the type of operating train. And the value of attribute service could assist in classification. But not all graphs have this attribute. Besides, we need to keep each two index-adjacent edges in every graph intersect in head-tail type, i.e., they have exactly one common end-vertex. Also all the edge-ids in a graph should be stored in topological order. Otherwise we will miss a part of information while transforming the network into a GTFS feed.

To solve these aforementioned problems, the network is improved in five incremental steps: Order Repair, Gap Repair, Classification, Connection Repair and
Topological Sort. One thing to note is the equal vertex identification is implanted in gap-check/connection-check of Order Repair, Gap Repair and Connection Repair parts.

![Equal Vertex Identification](image_url)

**Figure 4.2:** A picture of Equal Vertex Identification.

### 4.2.1 Order Repair

Normally most of the edges in graph are in position, only a few aren’t. Thus the major mission of this part is to get as many edges as possible in position, particularly the front end-edge.

The basic idea of Order Repair is a fake gap can be either removed or transformed into a real gap through bidirectional A* algorithm. The bidirectional A* algorithm we used is a little different from the normal one, such that it can tolerate the existence of a real gap in its results. In other words, the bidirectional A* algorithm can output a path with a real gap in it, if there isn’t one without a real gap. In this way all the edges in the found path are repetitive. But for the following two reasons we shouldn’t be concerned about a repetition of edge:

1. the repetitive edges wouldn’t result in a new fake gap on account of using directly conjoint edges as neighbors in bidirectional A* algorithm.
2. they can be removed through depth-first search.

In case of railway C from Figure 4.1, after Order Repair the order of edges in it should be

\[ w_1 \rightarrow w_4 \rightarrow w_3 \rightarrow w_2 \rightarrow w_4 \rightarrow w_3 \rightarrow w_2 \rightarrow w_5 \rightarrow w_6. \]

We can see that edges \( w_2, w_3 \) and \( w_4 \) are repeated. This example proves that with the help of the existence of repeated edges we can get a correct adjacency list for each edge through processing edges in order.

More important, the repeated edges enable us to find the front end-edge with high confidence. If the front end-edge is out of position, then after removing fake gaps the first not repetitious edge is the front end-edge with very high probability. The reason is that the front end-edge intersects with its neighbors at one identical conjoint point. For instance, in Figure 4.1 the front end-edge \( w_4 \) of railway C is out of position. After Order Repair the order of edges in it should be

\[ w_1 \rightarrow w_2 \rightarrow w_3 \rightarrow w_2 \rightarrow w_1 \rightarrow w_4 \rightarrow w_1 \rightarrow w_2 \rightarrow w_3 \rightarrow w_5 \rightarrow w_6. \]
4.2 Repair System

It’s obvious that edges $w_1$, $w_2$ and $w_3$ are repeated. Then we delete all edges in front of $w_4$, and the order becomes

$$w_4 \rightarrow w_1 \rightarrow w_2 \rightarrow w_3 \rightarrow w_5 \rightarrow w_6.$$ 

The front end-edge $w_4$ is finally in position.

In the following text we sketch the steps briefly.

At first, we check a graph to dig out gaps and record them.

More over, we use all the edges in this graph to build an R-Tree, such that given an edge we can find all its conjoint neighbors (edges).

In the next place, we find a path using bidirectional A* algorithm to fix each gap.

Moreover, we check every edge in order, how many duplicates it has. If we find the first edge, which doesn’t have any duplicates, we delete all the edges in front of it.

At last, we repeat above mentioned 4 steps for every graph.

4.2.2 Gap Repair

After Order Repair we only have real gaps left in network. Hence, this part devotes itself to repair as many as possible gaps to make network complete.

Firstly, each graph is checked for the existence of gap and we store those gaps in units of graph. Particularly, we only care about whether two index-adjacent edges are conjoint or not, their conjoint position is not our concern in this part.

Secondly, we build an R-Tree using every edge to find out all the conjoint neighbors (edges) of a given edge.

Eventually, we repair these gaps in each graph using a path finding algorithm. If a gap is unfixable, then we stop repairing this graph. Because those edges behind this gap are going to be deleted for keeping the graph’s connectivity.

![Figure 4.3: A picture of Gap Repair.]

The dotted lines $w_3$, $w_4$ in “before Gap-Repair” part mean they are missing.

4.2.3 Classification

In this part we focus on giving each graph an appropriate label: local or long-distance.
First of all, all the graphs are divided into three classes: local, long-distance and unknown with the aim of the feature service: given a graph

- if it has one of the attributes service=regional, service=tourism, service=commuter, service=commuter_rail, then it’s local.
- if one of the attributes service=long_distance, service=high_speed, service=night, service=express, service=national, service=international is in it, it’s long-distance.
- otherwise, it is unknown.

While grouping, we also need to calculate three values for each local or long-distance graph:

- **label**: means its class. -1 stands for local and 1 for long-distance.
- **length**: is computed by addition of the lengths of all the edges included in it.
- **score**: represents the alphabetical prefix of the value of attribute ref. If the graph doesn’t contain the attribute ref, its score is 0. If the prefix contains more than 6 characters, we use the maximum number to indicate it. In other cases we use a hash table, which contains 156 individual prime numbers, i.e. each character has 6 prime numbers and each one of them represents the index of this character in the prefix. In the calculation of score we multiply the appropriate prime numbers of the characters in prefix. For example, we have a graph with attribute ref=aa123. The prime numbers for ‘a’ in hash table is [2, 3, 5, 7, 11, 13], then its score is hash[a][0] * hash[a][1] = 2 * 3 = 6

We store them in form of a `trainingNode` in `trainingList`. `TrainingNode` is a simple structure, it contains only three variables: **label**, **length** and **score**. `TrainingList` is a list of `trainingNodes`. All the unknown graphs are saved in a list `unknownGraphs` to wait for being predicted later.

In the next place, we analyze the **score** of each **trainingNode** to discover the constant mapping relations between some particular values and classes. Due to using prime numbers in the calculation of score, every different value indicates one unique prefix of attribute ref. So in this step we actually search for the mapping relations between some prefixes and classes. As we know, these relations do exist. For example, the prefixes “re” and “en” are abbreviations for “regional express” and “Europe night line” respectively. Thus a graph with “re” is local and with “en” is long-distance.

Once again, we use these mapping relations to predict classes for unknown graphs. If we found a match, we delete it from the list `unknownGraphs` and insert a new `trainingNode` with its score, length and label into `trainingList`.

After that, we use SVM and the `trainingList` to build a predicting model. But only the **length** feature is used for predicting not **score**.

In the end, we use the obtained predicting mode to find a proper class for the rest graphs in `unknownGraphs`. 
4.2 Repair System

4.2.4 Connection Repair

Before the Topological Sort part, we need to make sure that there isn’t a gap in each graph and each pair of index-adjacent edges is conjoint in head-tail type. Therefore, the task of this part is removing gaps and repairing connections between edges.

For each graph, we check the connectivity between edges first. If two edges intersect, we need to ensure they’re conjoint in head-tail type. If need, we separate an edge to achieve it. When two edges are not adjacent, we delete the second edge and the ones behind it. Here we assign each edge in this graph a state to mark its connection situation.

- **normal**: The head end-vertex of this edge is the conjoint point with its front neighbor, and its tail end-vertex is one of the end-vertices of its back neighbor.

- **inverse**: its tail end-vertex is an end-vertex of its front neighbor and its head end-vertex belongs to its back neighbor also.

- **redundant-normal**: only its head end-vertex is in its front neighbor.

- **redundant-inverse**: only its tail end-vertex is in its front neighbor.

After repeating this step for every graph, we have accomplished the task of this part.

![Figure 4.4: A picture of Edge Separation in Connection Repair.](image)

The blue point means it’s not an end-vertex of $w_2$ but it’s an end-vertex of $w_3$.

4.2.5 Topological Sort

In this part we’re able to delete repetitive edges and organize the edges in every graph.

For a start, we create an adjacency list for each edge based on its state in a graph. Secondary, we apply DFS algorithm to the edge list of the graph. Eventually, we repeat the above two steps for all graphs, such that each one of them has a topologically sorted edge list.

So far, the Repair System chapter is finished. And the network is ready for the transformation.
4.3 Generating GTFS Feed

In chapter 3.1.4 we already introduced the GTFS system, so the major subject in this part is how to use it to transform a network into a GTFS feed.

Above all, we build an R-Tree with the coordinates of the vertexes, which represent a station. It’s used for the identification of same stations. Because we avoid getting ourself into the situation, that many different coordinates describe one same station.

In the next place, we do the transformation. Since we process every graph in network in the same way, we present the concrete steps only once.

– Firstly, we create a route with the attributes of the graph and add it to GTFS system.

– Secondly, the graph is separated into several railways, which indicate one of its branches.

– Thirdly, we create a trip and check each vertex in each railway’s edge whether it’s a station or not. A vertex is a station, if and only if it contains one of the attributes public_service=stop_position, public_service=stop_area, public_service=station, public_service=platform. After identifying a vertex as a station we check whether this vertex belongs to a big station or not. We first check if it contains the attribute uic_ref. If not, we use the R-Tree to find its nearest station and check if they both stand for a same station through comparing their distance with a threshold value. If they are, we consider two cases:

1. the nearest station has the attribute uic_ref. Then we assign this vertex its nearest station’s uic_ref attribute.

2. the nearest station doesn’t contain this attribute. In this case, we create a value for uic_ref and assign them uic_ref with it.

After all this, we create a stop with the vertex’s uic_ref (or id if uic_ref is not available), name and coordinate, and add it to GTFS system. Then we append this stop’s latitude and longitude together with a created fake arrival_time and a created fake departure_time to the current trip. The creation of fake arrival_time and departure_time follows the rules:

• railway network operates all day long.

• time interval between arrival_time and departure_time of each station is 180 seconds.

• the speed of local train is 160 km/h and long-distance is 200 km/h.

• the arrival_time of starting station of every trip is always on the hour.

• the train’s runtime between two stations is calculated by

\[ t = \frac{distance_A \text{ to } B}{speed}, \]
and the $distance_{A \to B}$ is the length of the route between $A$ and $B$.

No matter whether this vertex is a station or not, we add its coordinate to the current railway. When all the vertexes are examined, we create a shape with content of the current railway. And we generate 23 more trips by increasing each arrival_time and each departure_time of the prior created one 1 hour each time. These 24 trips indicate that the trains operates continuously in 24 hours. Then we insert the created trips and shape into GTFS system. At last, we output this GTFS system to the files included by a GTFS feed through a simple CSV writer designed by us.
5 Evaluation

The ExtractionFromOSM application was installed in a personal computer with an 2.6 GHz quad-core Intel Core i7 processor, 16 GB RAM, and a 64-bit operating system.

We ran ExtractionFromOSM on Germany OSM data, Europe OSM data and the whole planet OSM data. Figure 5.1 shows the running results of our program. Through two comparisons from the running results:

1. compare the runtime, number of successful repaired gaps between using A* algorithm and Dijkstra’s algorithm;
2. compare number of successful repaired gaps between with Order Repair and without it.

we arrive at two conclusions:

1. the difference between using A* algorithm and using Dijkstra’s algorithm is pretty small, almost the same;
2. Order Repair exerts an significant influence on removing gaps, i.e., after Order Repair most of the edges are in position.

Then we upload the GTFS feed of Europe to TRAVIC and OTP. The displayed results in TRAVIC prove that our method works perfectly.

In OTP the route-finding works perfectly fine for some routes (Figure 5.3). But for other routes route-finding doesn’t work and the error message “Trip is not possible. Your start or end point might not be safely accessible (for instance), you might be starting on a residential street connected only to a highway). (Error 404)” pops up.

The possible reason is that OTP doesn’t recognize the features location_type and parent_station in stops.txt. Normally in a large station there are many tracks. And each one of them has a stop inside this station. With the help of location_type and parent_station in stops.txt, we can use one of these stops, which belong to a same station, to represent this station. Since OTP doesn’t use them, each stop is treated as a station in OTP. In this case routing should work also, only a few meters walking-distance from one stop to the other would be displayed in OTP. However, OTP needs an extra OSM street data to do the routing by foot or bicycle. Considering the huge memory demand of OTP, it’s impossible to supply the OSM data for Europe with our GTFS feed together to OTP. Thus, the routing results of our GTFS feed in OTP are not too good.
**Figure 5.1**: Performance Table.

<table>
<thead>
<tr>
<th>Runtime (ms)</th>
<th>GERMANY</th>
<th>EUROPE</th>
<th>PLANET</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Graph Number</strong></td>
<td>2022</td>
<td>7959</td>
<td>12936</td>
</tr>
<tr>
<td><strong>Edge Number</strong></td>
<td>127244</td>
<td>439244</td>
<td>688197</td>
</tr>
<tr>
<td><strong>Vertex Number</strong></td>
<td>788464</td>
<td>4227864</td>
<td>8290301</td>
</tr>
<tr>
<td><strong>Data Extraction</strong></td>
<td>1262728</td>
<td>1186926</td>
<td>1246597</td>
</tr>
<tr>
<td><strong>Order Repair</strong></td>
<td>1967498</td>
<td>0</td>
<td>1951216</td>
</tr>
<tr>
<td><strong>Gap Repair</strong></td>
<td>683592</td>
<td>3557606</td>
<td>558782</td>
</tr>
<tr>
<td><strong>Classification</strong></td>
<td>1222</td>
<td>1212</td>
<td>1553</td>
</tr>
<tr>
<td><strong>Connection Repair</strong></td>
<td>1709</td>
<td>1615</td>
<td>2347</td>
</tr>
<tr>
<td><strong>Topological Sort</strong></td>
<td>330</td>
<td>309</td>
<td>368</td>
</tr>
<tr>
<td><strong>Generating GTFS Feed</strong></td>
<td>14907</td>
<td>17812</td>
<td>17359</td>
</tr>
<tr>
<td><strong>Gap Number</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Removed by OR</strong></td>
<td>9315</td>
<td>0</td>
<td>9315</td>
</tr>
<tr>
<td><strong>Removed by GR</strong></td>
<td>162</td>
<td>1955</td>
<td>163</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>23243</td>
<td>180886</td>
<td>235644</td>
</tr>
<tr>
<td><strong>Gap-repaired Rate</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Order Repair</strong></td>
<td>40,0 %</td>
<td>0,0 %</td>
<td>40,0 %</td>
</tr>
<tr>
<td><strong>Gap Repair</strong></td>
<td>0,7 %</td>
<td>8,4 %</td>
<td>0,7 %</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>40,7 %</td>
<td>8,4 %</td>
<td>40,7 %</td>
</tr>
</tbody>
</table>

* O.R. is ab. for Order Repair.
Figure 5.2: Pictures of display results from TRAVIC.
Figure 5.3: Pictures of display results from OTP.
6 Further Research

6.1 Extending network by collecting all the ways near to at least one of the railways

In this thesis only all the train-related ways are stored in form of edge in network. But still a lot of gaps are not fixable in the lack of necessary ways. Instead of loading all the ways in network, only collecting the ones near to railway can reduce runtime of the program without decreasing its efficiency. To do the information collection we need to follow three steps:

1. we build an R-Tree using the geographical lines presented by all the edges in network.
2. with the help of an R-Tree we find the nodes from those ones excluded by network, which is near to at least one of railways in a certain distance. And each found node is transformed into a vertex in network.
3. we search the ways only containing the vertexes included by network from those network-excluded ways. Then we load them in network.

In the program this method is already implemented.

6.2 Improving the efficiency of classification

In this paper the Support Vector Machine is used for the 2-class classification. Due to the existence of some long-distance graphs with short lengths there are biases in our predicting result. So if we process the training data first, such as finding a low length-boundary for long-distance class to eliminate the influence of those graphs, the predicting result may be much more reasonable.

6.3 Identifying neighbors with regard to the existence of a real gap

In the Order Repair part our method takes only the directly conjoint neighbors into account. So if we meet an extreme case that source edge and target edge are
isolated, i.e., they don’t have any conjoint neighbors, then our method wouldn’t work. But if our method can find those neighbors through a real gap, the efficiency of Order Repair may be enhanced.
Bibliography


