Undergraduate's Thesis

Merging of Overlapping GTFS Feeds

Leo Felix Zeches

Examiner: Prof. Dr. Hannah Bast
Advisers: Patrick Brosi

Albert-Ludwigs-University Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Algorithms and Data Structures

April 04th, 2019

Writing Period

08. 01. 2019 – 08. 04. 2019

Examiner

Prof. Dr. Hannah Bast

Advisers

Patrick Brosi

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

The General Transit Feed Specification, GTFS for short, is a commonly used format for public transportation schedules and related geographical information. Sometimes, these feeds are covering the same network of public transportation and can overlap for certain stations, routes and trips. This undergraduate thesis describes a way to merge such trips in a manner that retains all the relevant information, as well as recognizing duplicate trips and merging trips that are partially included in each other.

Zusammenfassung

"General Transit Feed Specification", kurz GTFS, ist ein verbreitetes Format welches für Fahrpläne im Bereich der öffentlichen Verkehrsmittel benutzt wird. Teilweise kommt es vor, dass solche Feeds das gleiche Netzwerk abdecken und sich so für Stationen, Routen oder Trips überschneiden. Diese Bachelorarbeit beschreibt eine Möglichkeit, solche Feeds zu erkennen und miteinander zu vereinigen. Dabei werden alle relevanten Informationen erhalten sowie Duplikate und Überschneidungen erkannt und angemessen behandelt.

Contents

List of Figures

List of Tables

1 Introduction

The organization and managing of public transportation is a well documented and prevailing field of work. As technology and connectivity increases, so does the need for managing and organizing the accompanying data sets. In the field of public transportation, many companies use the *General Transit Feed Specification*, or GTFS for short. It contains relevant information regarding trips, stops and routes. This common format is easy to create, read and use for a multitude of applications, which is why many companies chose to publish their GTFS feeds to be used by developers for a range of purposes. This thesis deals with the merging of two overlapping GTFS feeds.

Different traffic companies tend to create their own GTFS feeds to fit their individual needs. A feed from the company "*Deutsche Bahn*" can look very different to a feed from the Swiss company "*VÖV UTP*".

Problems usually arise when two companies cover the same area or an overlapping area. Take for example the public transportation between Germany and Switzerland. Figure 1 shows the different public transport companies for the different regions in Germany, Austria and Switzerland. Figure 2 shows different IC and EC routes from Germany to other countries, such as Austria or Switzerland. On figure 3, we see the IC plan for Switzerland. A German public transport company may choose to include all of the international stops and trips in its GTFS feed. A Swiss company may chose to only include local trips.

On listing 1.1 and listing 1.2, we can see a simple example for how two trips from different agencies could look like. On listing 1.1, we have a trip "FBK" starting in Freiburg and proceeding to Basel and Zürich. On listing 1.2, we have a trip "BaZue" from a different public transportation agency. This trip is starting in Basel and then proceeds to Zürich. Since their arrival- and departure times and stops are the same, it is reasonable to suspect that the trip "BaZue" in feed 2 is included in the trip "FBK" from feed 1.

```
trip_id,arrival_time,departure_time,stop_id,stop_sequence
FBK,8:00:00,8:05:00,FreiburgHBF,1
FBK,8:45:00,8:50:00,BaselHBF,2
FBK,9:50:00,10:00:00,ZuerichHBF,3
```

Listing 1.1: Example for *stop_times.txt* for a GTFS feed *A*

```
trip_id,arrival_time,departure_time,stop_id,stop_sequence
BaZue,8:45:00,8:50:00,Basel,1
BaZue,9:50:00,10:00:00,Zuerich,2
```

Listing 1.2: Example for *stop_times.txt* for a GTFS feed *B*

If we want to merge those two GTFS feeds, we can end up with a lot of redundant information, multiple equivalent router and overlapping trips. For that reason, we need to find a way to compare different trips and merge them, if they are at the same stops at the same times. This is not as simple as it may seem at first; different GTFS feeds may have very different values for the same stops or trips. For example, a stop could be marked as "Freiburg Hauptbahnhof" or as "Freiburg im Breisgau, HBF".
In this thesis, we will discuss an approach to merge overlapping GTFS feeds, using geological location and stop times to match and merge trips. Our output is a valid GTFS feed with no redundant information that can be used for other applications.

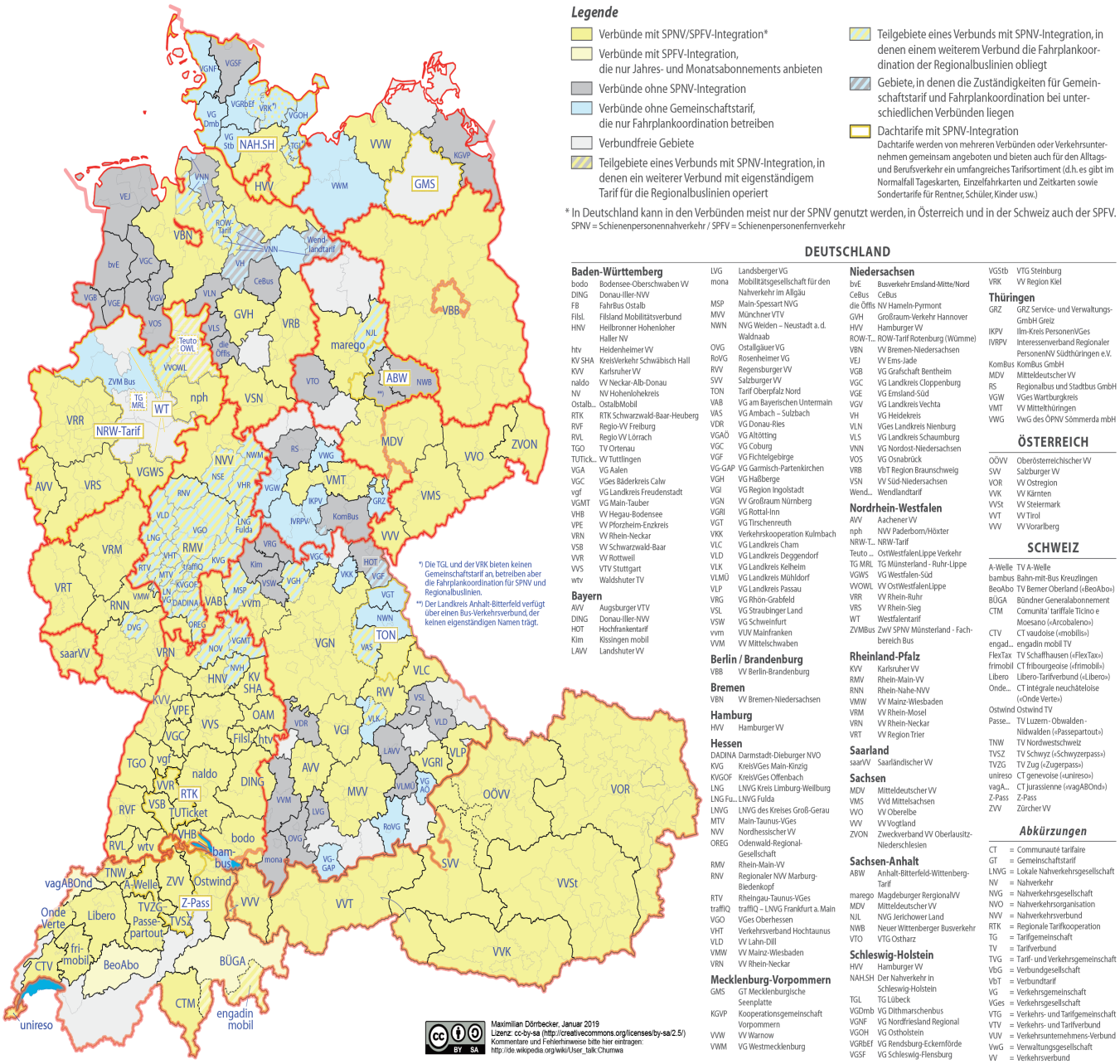

Figure 1: Public Transport Organizations in Germany, Austria and Switzerland, 2019

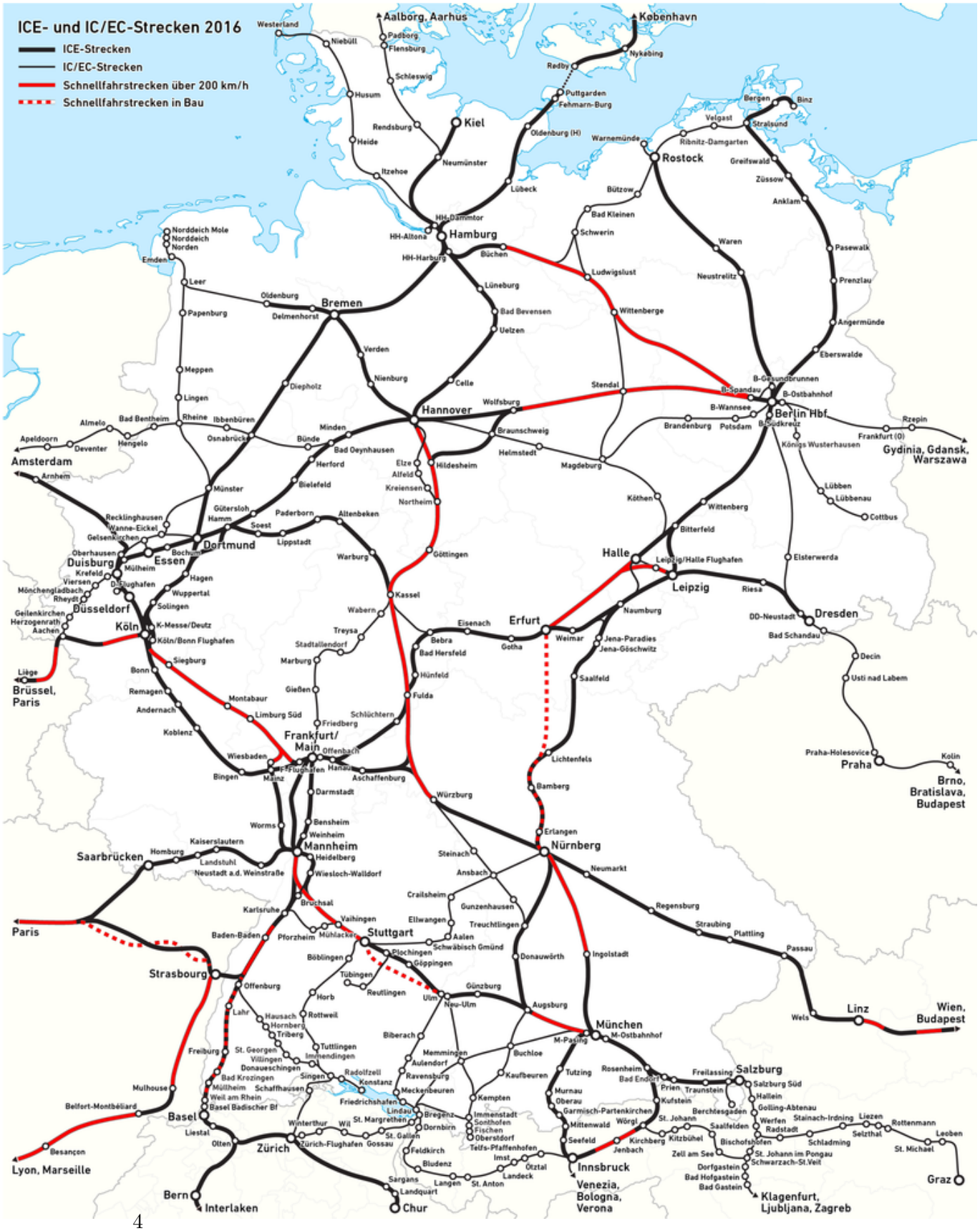

Figure 2: IC and EC routes in Germany, 2016

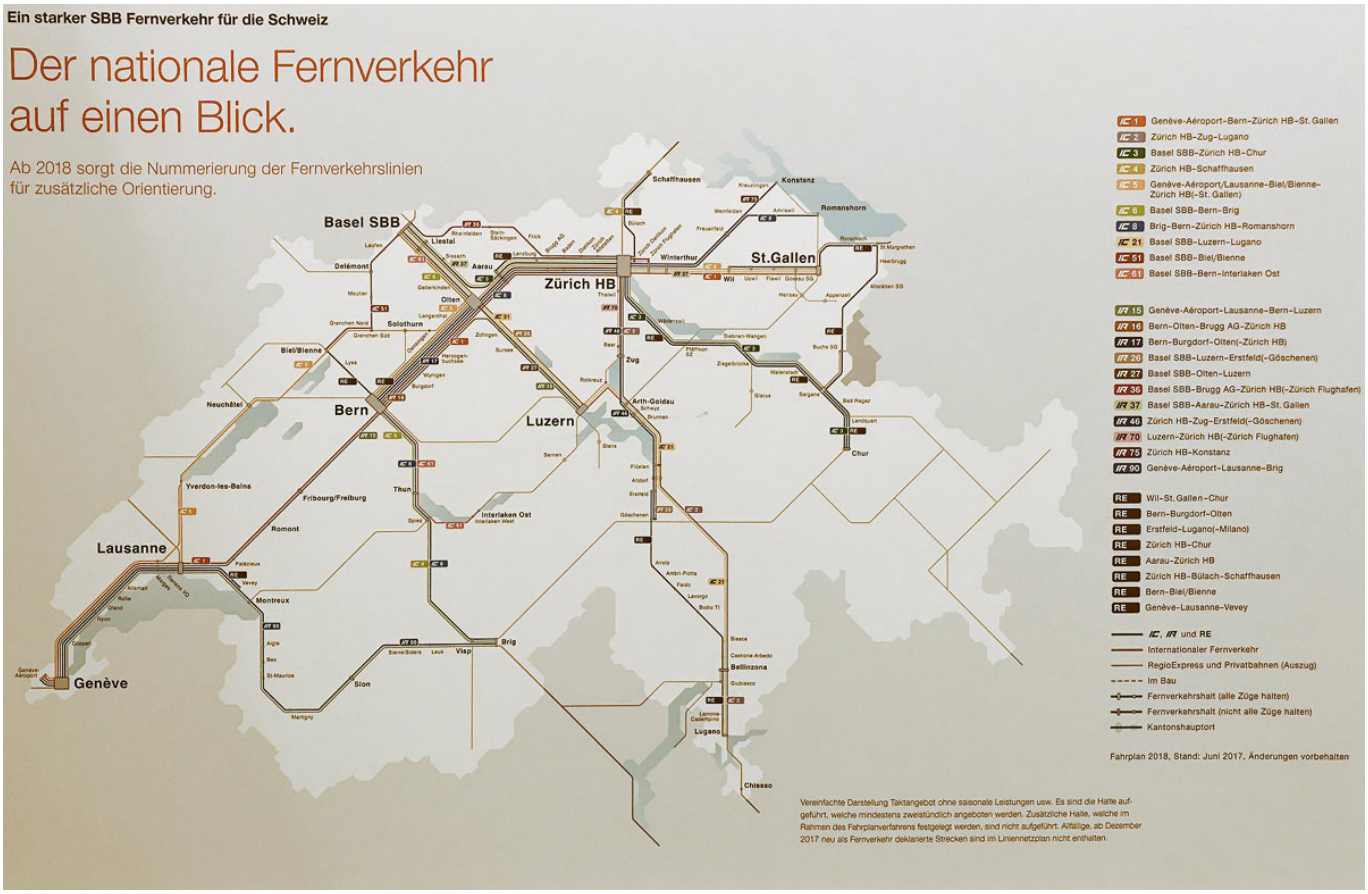

Figure 3: IC routes in Switzerland, 2018

2 Related Work

After extensive research, we could not find any relevant work that focused on merging overlapping GTFS feeds. In this chapter, we will have a look at a few thesis' and works that revolve around GTFS feeds in general.

On Github, a merging tool simply titled "Merge" [1] exists. This tool is used if an agency changes their service periods and wants to to merge existing trips to close gaps in their service times. According to the documentaion, this tool is mainly intended to merge schedules. New stops will be handled but other changes may cause problems.

In 2017, Tran et al. [2] published a thesis on semantic comparisons between GTFS feeds. In this thesis, the contents of trips are compared to each other on a semantic level. After reading in the feeds, the program starts by comparing the different routes. For this comparison, the entries for *route_type, route_short_name* and *route_long_name* are compared to each other using the levenshtein algorithm to allow for small deviations. Then, trips that are stored in hash maps using *stop_times* entries as keys. When comparing the trips to each other, the results are stored as *TripData-Objects* that contain additional information like the percentage of differences in the stop_times. Stops are compared if two *stop_time* objects have the same *arrival_time* and *departure_time*. Stop objects are matched using their *stop_names* and the levenshtein algorithm as well as their geographical location using the Vincenty formula. Finally, the service times are compared using the week days and *start_dates* and *end_dates*.

While this work focuses on finding semantic differences and similarities between

GTFS feeds, our thesis focuses on merging overlapping GTFS feeds.

In 2017, Zhang et al. [3] published a thesis on public-transit data extraction from OpenStreetMap (OSM for short). In the first step, the relevant information is extracted from the OSM data and stored in a railway network. The second step focuses on fixing incorrect entries, such as missing information or gaps. Finally, the network is transformed into a valid GTFS feed and visualized using Transit Visualization Client (TVC) and OpenTripPlanner (OTP). The paper also talks about the limitations of OSM data and the resulting performance.

3 Theoretical Background

We used a number of data structures and algorithms in the implementation of our program. In the following chapter, we will take a look at the most important ones, as well as giving a small overview of the most important external libraries used.

3.1 Hash Maps

A hash map is used to store data elements using a key. This key can then be used to search and access the element. The key is calculated using a *hash function*. A hash function usually takes the data to be stored and calculates a key based on that data itself. This key is used to access the data later on. In our program, we use the python hash function to calculate a key to store our data into a hash map.
In our program, mostly use python dictionaries, which is a form of a hash map. The keys are set manually, so that we know the value of the key. When using dictionaries and hash maps, we have to make sure our keys are well distributed in order to avoid collisions.
A useful aspect of hash maps is the efficient running time to access a data element. If we know the key of the element, we can access it in $O(1)$ if the hash map is constructed well and if the elements are distributed evenly.

3.2 GTFS-Format

The General Transit Feed Specification [4] is a format developed by Google. It is made up of text files usually compressed into a zip file. The documentation specifies

13 text files containing CSV-data (comma-separated-values), with 7 of these text files being required, and the other 6 being optional. The files themselves have required and optional attributes. In this thesis, we will only work with the required files and attributes, which are as follows:

- agency.txt: One or multiple transit agencies that provide the data for the GTFS feed. Contains the required attributes *agency_name, agency_url* and *agency_timezone*.
- stops.txt: The individual stop locations. Contains the required attributes *stop_id, stop_name, stop_lat* and *stop_lon*.
- routes.txt: The transit routes. A route is a group of trips. Contains the required attributes *route_id, route_short_name, route_long_name* and *route_type*.
- trips.txt The trips for each route. A trip is a sequence of multiple stops for a specific time. Contains the required attributes *route_id, service_id* and *trip_id*.
- stop_times.txt The different times for when a stop is frequented. Contains the required attributes *trip_id, arrival_time, departure_time, stop_id* and *stop_sequence*.
- calendar.txt Dates for services with a weekly schedule. Contains the required attributes *service_id, monday, tuesday, wednesday, thursday, friday, saturday, sunday, start_date* and *end_date*.
- calendar_dates.txt Exceptions for the dates specified in calendar.txt. Contains the required attributes *service_id, date* and *exception_type*.

On listing 3.1, we see a section the stop_times.txt file from the example file *sample-feed.zip* from the Google developer site.

```
trip_id,arrival_time,departure_time,stop_id,stop_sequence,
stop_headsign,pickup_type,drop_off_time,shape_dist_traveled
STBA,6:00:00,6:00:00,STAGECOACH,1,,,,
STBA,6:20:00,6:20:00,BEATTY_AIRPORT,2,,,,
CITY1,6:00:00,6:00:00,STAGECOACH,1,,,,
CITY1,6:05:00,6:07:00,NANAA,2,,,,
CITY1,6:12:00,6:14:00,NADAV,3,,,,
CITY1,6:19:00,6:21:00,DADAN,4,,,,
CITY1,6:26:00,6:28:00,EMSI,5,,,,
CITY2,6:28:00,6:30:00,EMSI,1,,,,
CITY2,6:35:00,6:37:00,DADAN,2,,,,
CITY2,6:42:00,6:44:00,NADAV,3,,,,
CITY2,6:49:00,6:51:00,NANAA,4,,,,
CITY2,6:56:00,6:58:00,STAGECOACH,5,,,,
AB1,8:00:00,8:00:00,BEATTY_AIRPORT,1,,,,
AB1,8:10:00,8:15:00,BULLFROG,2,,,,
AB2,12:05:00,12:05:00,BULLFROG,1,,,,
AB2,12:15:00,12:15:00,BEATTY_AIRPORT,2,,,,
BFC1,8:20:00,8:20:00,BULLFROG,1,,,,
BFC1,9:20:00,9:20:00,FUR_CREEK_RES,2,,,,
BFC2,11:00:00,11:00:00,FUR_CREEK_RES,1,,,,
BFC2,12:00:00,12:00:00,BULLFROG,2,,,,
```

Listing 3.1: Section of *stop_times.txt* from sample-feed.zip

3.3 Data Structures

Our program is based on the concept of object-oriented programming. In order to handle the data effectively, we created an object class for every kind of text file in

the GTFS feed. Using these classes, we create objects for every entry of a text file. The object classes are as follows:

- Stop:

  Contains variables for the required and optional attributes of an entry in stops.txt. Additionally it has a method to change *stop_name* as well as *stop_lat* and *stop_lon*. This method is used for our evaluation, which will be explained in detail in chapter 5.1.

- StopTime

  Contains variables for the required and optional attributes of an entry in stop_times.txt. Additionally it has a method to change the values of *arrival_time* and *departure_time*. This method is used for our evaluation, which will be explained in detail in chapter 5.1.

- Route

  Contains variables for the required and optional attributes of an entry in routes.txt.

- Agency

  Contains variables for the required and optional attributes of an entry in agency.txt.

- Calendar

  Contains variables for the required and optional attributes of an entry in calendar.txt.

- CalendarDates

  Contains variables for the required and optional attributes of an entry in calendar_dates.txt.

- Trip

  Contains variables for the required and optional attributes of an entry in trips.txt. Additionally, this class contains list variables to store other objects. Each trip object has a list for *stop_times, stops, routes, calendars, calendar_dates, routes* and *agencies*. These are filled with the objects that are referencing the trip object itself. This way, a trip object contains all of the other objects relevant to itself. A trip object also has a method to change the *trip_id* and a method to create noise in other objects. These methods are used for our evaluation, which will be explained in detail in chapter 5.1. A trip object also contains a ___key___ attribute with a hash value we use by hashing the entries for *service_id, route_id* and *trip_id*. A trip object also contains a set ___compared___, used to store the trip keys of every trip we already compared. This allows us to avoid comparing the same trips multiple times.

3.4 2D Grid

To optimize the running time of our program, we construct a 2D grid using python lists. A 2D grid is one large list where each element is another list. This creates a data structure where we can store and access elements using two index, one for the outer list (or x-axis) and one for the inner list (y-axis).
The grid is used to store keys of trips that have a similar geographical location. Each field of the grid represents a 20km by 20km area. For each of our stop objects, we calculate the distance to our starting point (0,0) using the haversine formula. This distance is then divided by 20, giving us the index for the field in which the stop is located. Then, every trip that accesses this stop is written into the field.
When comparing our trips, we only compare the trips that are in the same field or in the adjacent fields. This avoids comparing trips that share no stops among them, reducing the number of trips we need to compare with each other. This greatly

improves our running time.

3.5 Different Types of Trips

In our program, we check the trips for 4 different cases. These cases are as follows:

- Case 1: Two trips are equivalent

  For two trips to be equivalent, they must frequent the same stops at the same time at the same dates. Only if every stop and time matches for both trips, we can make sure that they are equivalent and delete one of the two trips from the merged feed.

- Case 2: A small trip is included in a bigger trip

  This is the case when a first trip has less stop_times than a second trip. Besides that, the smaller trip needs to frequent the same stops at the same times at the same dates for every stop_time. If this is the case, we can conclude that the smaller trip is included in the bigger trip and delete the smaller trip from the merged feed.

- Case 3: Two trips are partially included in each other

  If the end of *tripA* and the start of *tripB* are overlapping, then they are partially included in each other. If this is the case, we create a new trip out of them, starting from the beginning of *tripA*, continuing over the intersecting part of both trips, and ends on the remaining trips from *tripB*. We append the new trip to our merged feed and delete *tripA* and *tripB*.

- Case 4: Two trips are distinct

  If none the cases 1-3 applies, the trips are completely distinct and we keep them both in our merged feed.

3.6 Algorithms

3.6.1 Haversine Formula

The haversine formula determines the *great-circle-distance* between two points on a spherical model of the earth, given their latitudes and longitudes. We use this formula in our program to determine the distance between two stop objects. The formula looks as follows:
Let the central angle θ between two points on a sphere be defined as:

$$\theta = \frac{d}{r}$$

where d is the distance between the two points and r is the radius of the sphere.
The haversine formula of Θ is given by:

$$hav(\Theta) = hav(\varphi_2 - \varphi_1) + \cos(\varphi_1) \cdot \cos(\varphi_2) \cdot hav(\lambda_2 - \lambda_1)$$

where φ_1, φ_2 are the latitudes of point 1 and point 2 and λ_1, λ_2 are the longitudes of point 1 and point 2.
The haversine function of an angle θ is:

$$hav(\theta) = \sin^2(\frac{\theta}{2}) = \frac{1-\cos(\theta)}{2}$$

In order to solve for distance d, we apply the inverse function to the central angle Θ:

$$d = r \cdot hav^{-1}(hav(\Theta)) = 2 \cdot r \cdot \arcsin(\sqrt{hav(\Theta)})$$

or, more explicitly:

$$d = 2 \cdot r \cdot \arcsin(\sqrt{\sin^2(\frac{\varphi_2-\varphi_1}{2}) + \cos(\varphi_1) \cdot \cos(\varphi_2) \cdot \sin^2(\frac{\lambda_2-\lambda_1}{2})})$$

As mentioned before, the haversine formula calculates the distance between two points on a perfect sphere. This means that the results will always be a (very close) approximation, since we know that the earth is not a perfect sphere. To calculate the distance between two points on the surface of a spheroid, we would have to

use the *vincenty formula* [5]. This would provide us with an even more accurate distance between two stops. However, we chose not to use this formula since the calculation of the vincenty formula is iterative and uses a lot more computing power. Using the haversine formula provides us with a result that is accurate enough to determine if two stop points are on the same geographical location, while not using up unnecessary computing power.

3.6.2 Timsort

To sort lists in our programs, we use the python built-in function *sort()*, which is using a sorting algorithm called *Timsort* [6]. Timsort works as follows:
If the list has fewer than 64 items, Timsort will execute an *insertion sort*. If the list has more than 64 items, which is mostly the case in our program, Timsort will pass through the list, checking for parts that are already strictly increasing or decreasing. If parts are decreasing, those parts are switched. Then the algorithm divides the list up into block known as *Run*. The size of run varies between 32 and 64, depending on the size of the list. The run blocks are sorted using insertion sort and then merging the run blocks using merge sort.
The time complexity of Timsort is $O(n)$ in the best case and $O(n \cdot \log(n))$ in the average and worst case.

3.7 Memory Usage

The majority of the information handled by our program comes in the form of strings. These are stored into our created objects. The largest amount of data we have to handle are the hash keys in the trip object set *___compared___*. This set contains all of the hash keys for the trips already compared to the trip in question. This means that we have a potential of $(m + n)^2$ entries for the combined sets in the worst case, with n being the number of trips in feed1 and m being the number of trips in feed2. The size of these keys is *size_t*, which is 64 bit or 8 byte on a 64

bit machine. This has the potential to increase memory usage for big data sets. Imagine two GTFS feeds with 10'000 trips each. This means that we have to store $20'000^2 = 400'000'000$ hash keys in the worst case. At 8 bytes a key, this equals 3.2 GB of memory storage for the hash keys. While this is an amount most machines can handle, it is understandable how this can quickly become an issue when dealing with greater amounts of data. In the average case, we have to only This point is further addressed in chapter 6.

3.8 External Libraries

- csv

  CSV is a python library to read and write csv files. Since we are dealing exclusively with csv text files in our GTFS feeds, this tool allows us to easily read in our text files and create our objects. The first line of a file is taken as the header, containing the field names of the entries. Then, every line after the header is split at the comma symbols. Then every entry is mapped to its corresponding field name from the header. [7]

- haversine

  A simple library that uses the haversine formula to calculate the distance between two points on the Earth using their latitude and longitude. Returns the distance in miles or kilometers. [8]

- datetime

  The datetime module supplies classes for handling and manipulating dates and times. We mostly use the datetime.time and datetime.datetime classes, to make comparisons between times and dates easier. The format is [*hour, minute, second, microsecond*] for datetime.time and [*year, month, day, hour, minute, second, microsecond*] for datetime.datetime. [9]

- memory_profiler

This is a python module to monitor memory consumption of a process, or for a line by line analysis. In our case, we use the function decorator to analyze the memory consumption of our main function in *gtfsMerge*. [10]

4 Implementation

In the following section, we will take a look at the implementation of the program. The program can be split into 3 main parts:

1. Reading the GTFS-Feeds
2. Comparing and merging the trips
3. Creating the output

4.1 Reading in the GTFS-Feed

In this first part of the program, the two zip files are assigned and opened to access the different text files. Every text file we read has a header, which is the first line in the file. It provides the field names for the values that follow.

First, we read the *trips.txt* file from each of our two feeds, using our function *readTrips()*. The bytes are transformed into strings using TextIOWrapper [11]. Since we are dealing with comma-separated-values, we use the CSV-Reader to read the string. This gives us ordered dictionaries where the field names are the keys, and the entries are the values. These dictionaries are then used as input to create the trip objects.

The trip objects are stored in a dictionary using their *trip_id* as keys. We then read in each of the required text files, using the same approach: reading the entries, creating objects, storing them in a dictionary.

After reading every value of every required file in the two feeds, we start filling the trip objects using the function *fillTrips()*. The trip objects are filled with the corresponding *stop objects, stop_time objects list, calendar objects, calendar_date*

objects and *route objects*. This is done by comparing the dictionary keys with the values of the trips. Now every trip object has a list with the corresponding stop objects, stop_time objects, calendar objects, calendar_dates objects and route objects. The stop_data list is also sorted in order. This is so that we can iterate over them and check for matching sequences when comparing two trips.
Additionally, we create *stop_data tuples*. These tuples include a datetime object of the arrival time of a stop time object, as well as a stop object with the corresponding stop object. These stop_data tuples allow for an easier comparison between trips, providing us with a time and a place to compare. The tuples are stored in a list and added to the trip objects. Now our trip objects are ready to be compared and merged.

4.2 Comparing and Merging the Trips

In the following section we have a look at the main comparing function compareTrips() of our program.
We iterate over a dictionary containing our merged stop objects of feed 1 and feed 2. For every stop, we first take the trips that are in the grid field corresponding to this stop. Then, we also take the trips of the fields adjacent to the first field. This gives us all of the trips with a similar geographical location we need to compare. This list of trips is then sorted by their stop_times, to make that they are in order when we iterate over the trips. To sort the list, we use the python function sort().
We pop the first item from the list as *tripA*. Then we compare this first *tripA* to every other *tripB* in the remaining list and check if the trips fall into one of the four cases we referred to in chapter 3. We also add the keys of *tripA* to the *___compared___ set* of *tripB*, so we won't need to compare the same trips twice.
After iterating through the list once, we pop the next item and repeat the process again. This is done until every trip has been compared and we move on to the next stop.
Since we need to compare every trip with every other trip in the grid fields, our

comparison function is in $O((n+m)^2)$ in the worst case, with n being the number of trips in feed1 and m the number of trips in feed2. This is very rare in practice however, and our average case is $O(g^2)$, with g being the number of trips with a similar geographical location. This is still not perfect, and the problem of running time optimization will be discussed in further detail in chapter 6.
The comparisons are using the functions compareStopData() and compareCalendars() to compare stop_data tuples and calendar objects.

4.2.1 compareStopData

Comparing the stop_data tuples is done by comparing the arrival times and the stop objects separately. To compare the arrival times, we subtract one datetime object from the other and take the absolute difference in unit of seconds. We then divide this difference by 60, giving us the difference in units of minutes. We check if the difference is smaller than 3. If the difference is equal or greater than 3, we return *False*. We also compare the stop_objects by calculating their distance to each other. This is done by taking their coordinates *stop_lat* for the latitude and *stop_lon* for the longitude and calculating the distance using the haversine formula. This returns the distance in unit of kilometers. The distance is then multiplied by 1000, giving us the distance in units of meters. We check if the difference is smaller than 5. If the difference is equal or greater than 5, we return False.
If both comparisons result true, we conclude that both stop_data tuples are equal and return *True*. In essence, this means that two trips arrive at this same stop at the same time.

4.2.2 compareCalendars

Comparing the calendar objects is done by checking if their day lists are equal and if their start and stop date are equal. If there are differences, we conclude that the calendars describe different days and/or dates and we return *False*. If there are no differences, we conclude that the calendar objects describe the same time period,

and we return *True*.
This comparison does not handle overlapping start and end dates. This is a case that will be discussed further in chapter 6.

4.2.3 Case 1: equivalentTrips()

To check if two trips are equivalent, we use the *equivalentTrips()* function. We start by making sure that their stop_data lists have the same length. If they are not the same length, we already know they cannot be the same trip and continue with the next case check. If the lists have the same length, we compare their calendar lists and check if the trips are frequenting on the same week days. If this the case, we iterate once though both stop_data lists comparing the stop data tuples at the same index. Since they are sorted, this iteration will return *True* only if the sequence of stop data tuples (and therefor frequented stops) are equal.
If every iteration of stop tuples returns *True*, we conclude that the two trips are passing through the same stops at the same time for the same dates. We can then conclude that the trips are equal, even if their other values may differ from each other. If this is the case, we delete the trip from the second feed. Since the two trips are equal, we can merge them by deleting one of them. Then we continue with the next inner loop iteration over the trip list. However, if one pair of stop_data tuples is different, we break and check for the next case.

4.2.4 Case 2: includedTrips()

To check if one trip is included in the other trip, we start by checking if one trip has a smaller stop_data list than the other. Having this information, we can start the *includedTrips()* function. Again, we start by comparing the calendars of the two trips. If they are equal, we will continue by comparing the actual stop data tuples. Since the stop data tuples are in order, we check for a matching tuple and then iterate over the two lists at the same time, checking if the following stop data tuples match as well. If this is the case for the entire list of the smaller trip, we can conclude that

that trip is completely included in the bigger trip, even if their other values may differ. In that case, we delete the smaller trip and continue our iteration over the trip list. If not, we continue by checking for the third case.

4.2.5 Case 3: partiallyIncluded()

As in the previous cases, we check if the calendars are describing the same dates. If this is the case, we check if either the first stop_data tuple of the first trip matches a tuple in the second trip. If we have a match, we start checking if the following stop data tuples are matching as well. As soon as we find a unmatching pair after our first match, we can conclude that the trips are not partially included in each other, and we break the loop and exit the function.
If we do not find a match for the first tuple of the first trip, we check for the first tuple of the second trip. In order for two trips to be partially equivalent, at least one of the two first tuples of the two trips must find a match. If not, we can conclude that they are not partially equivalent, and do not need to compare the rest of the stop-data tuples. We can break the function after $2 \cdot n$ comparisons, where $n =$ number of stop_data tuples, because it takes us n comparisons to check if the first tuple of the first trip finds a match, and n comparisons to check if the first tuple fo the second trip finds a match.
Lastly, if we were able to find a match and iterate through the remaining tuples without a problem, we check if the number of common stop data tuples is greater than 1. This condition is important, because if only one stop data is equal in both trips, we can not definitely say that both trips can be merged. It could be the case that one trip is ending at the matching stop, and the other trip (on a different vehicle or on a different platform).
If we have a matching section of stop data tuples, we create a new trip object containing the whole sequence of stops and stop times. Then we delete both trips from their lists and append the new trip to the second list.

4.2.6 Merging two Trips and creating a new one

To merge two trips and create a new trip, we start by creating a new trip object taking the values for *route_id, service_id, trip_headsign, trip_short_name, direction_id, block_id, wheelchair_accessible, bikes_allowed* from one of the two trips that will be merged. For a new *trip_id*, we simply take the two values from the two merged trips and concatenate the two strings with a "+" in between. This is to make sure that we won't accidentally create a trip id that is already taken. (It is not impossible that an existing trip id is accidentally created, but using the "+" ensures that it won't happen in practice.) The value for *shape_id* will be set to "", because we cannot simply take one of the shape_ids from one trip. Creating a new shape_id will be discussed further in the chapter 6.
For the values of stop_data, we take the common stop_data of both trips and add the additional stop_data from trip 1 and from trip 2. The same is done to the values of stop_times, this time by only adding the additional values from trip 1 to the list of trip 2. Afterwards, we sort them by their arrival_time and redo their *stop_sequence* values. The stop values are gathered by extracting the stop objects from the new stop_data list. The stop objects themselves remain the same. Finally, we add the lists for calendars, calendar_dates, routes and agency from one of the two trips to our new trip object. This is simple, because if we merge two trips, their calendars, calendar_dares, routes and agency should be the same, so we can simply take over these values. Thus, our new trip object is finished.

4.3 Creating the Output

Once we have merged our trip dictionaries, we return them as a list of trip objects. Using this list, we can go through every trip and extract every *stop_time object, stop object, calendar object, calendar_dates object, route object* and *agency object*. This way, we have only the relevant objects for our newly merged trips. This also means that every object that did not belong to a trip in the original two feeds will be "lost".

This problem is addressed further in Chapter 6.

After extracting all of the values, we start creating the text files. Each file starts by the appropriate header. After that, we use the ___toString___() method of every object to fill in the entries. After creating every file, we zip them up using shutil.make_archive() function [12]. Our final output is a zip file called *"MergedFeed.zip"*, containing a trips.txt, stops.txt, stop_times.txt, calendars.txt, calendar_dates.txt, routes.txt and agency.txt file that contain the entries of the two merged feeds.

5 Evaluation

In order to evaluate our program, we merge different GTFS-Feeds and check if their essential information is retained, while the equivalent or overlapping feeds are handled in the right way. To achieve this, we have to create a ground truth. However, it would be impossible to create a real ground truth by merging and evaluating two real life feeds. We would have to analyze the newly created feed by hand, analyzing every line of every text file and comparing it to the two lines from the two original feeds. To avoid this problem, we can generate a "fake" ground truth by splitting an existing GTFS-Feed and merging it again. Then, we can compare the merged feed with the original feed and check if the program is working as intended.

5.1 The Evaluation Algorithm

To split a GTFS-Feed and compare it to our newly merged feed, we have created the Evaluation Algorithm.

5.1.1 Splitting a GTFS-Feed

We can use our algorithm to split an existing GTFS-Feed. This is done by firstly reading the contents of the zip-file using the same approach as our gtfsMerge algorithm. The trip dictionary is then split into two lists *tripsA* and *tripsB*. Random integers are used to determine whether a trip is put into tripsA or tripsB. Each trip has a chance of 40% chance to be put in either tripsA or tripsB. Each trip also has a 20% chance of being split into two different trips. To split a original trip into two new trips, we build two new trip objects. The values for *route_id, service_id, trip_headsign,*

direction_id, block_id are kept the same, since they do not need to change to split the trip effectively. The lists for *agency, calendar_dates, calendar* and *routes* are also kept the same for both trip objects. The first value we change is the *trip_id*, since this id should still be unique for this trip. We create two trip ids by splitting the id of the original trip and assigning each half to one of our newly created trips. We then select a split point for the *stop_time list*, since this is one of the lists we have to split to create two distinct trips. If the number of stop times is greater than 4, we split them in a way that they overlap each other (for example a trip with the stop times *"ABCD"* will be split into two trips with the stop times *"ABC"* and *"BCD"*). This way, we can evaluate the function of our program to recognize and merge partially overlapping trips. If the number of stop times is smaller than 4, we will split the stop times in a way that one smaller trip is included in a bigger trip (same example: trip with stop times *"ABC"* will be split into two trips with stop times *"AB"* and *"ABC"*). This way, we can also evaluate the function of our program to recognize and merge trips that are included in each other. We then rearrange the *stop_sequence* of the stop times so they are in order again, and give each stop time their new trip_id. After that, we assign the correct stop objects to each trip, since not every stop should be assigned to the newly created trips. Finally, each trip is added to either *tripsA* or *tripsB*. After every trip has been dealt with, we generate two GTFS-Feeds using our two lists of trip objects using the same approach as our main *gtfsMerge* program, and the final output is two zip files *SplitFeedA.zip* and *SplitFeedB.zip*.

5.1.2 Creating noise while splitting a GTFS-Feed

Another useful feature to evaluate the quality of our main algorithm is the creation of noise during the splitting process. The term *"noise"* describes changes in the original values of the split feed that makes it harder to match and merge trips. Noise mimics real life problems like spelling errors or small differences in *arrival_time* or *departure_time*. In our evaluation algorithm, we implemented 3 modes to add

different levels of noise:

Mode 0: no noise

The values of the trips are kept the same (except from the trip_id in the split trips)

Mode 1: small amount of noise

In this mode, we add small amount of noise. With a chance of 30%, a trip will add small changes, such as mixing up the *stop_name* for the stop objects or increasing the *arrival_time* and *departure_time* by a few seconds (max. 2 minute difference).

Mode 2: large amount of noise

In this mode, we add a large amount of noise. With a chance of 80%, a trip will add big changes. These changes include the same alterations as mentioned in mode 1. Additionally, the *arrival_time* and *departure_time* can be changed by up to 10 minutes. Additionally, we change important values such as trip_id and stop coordinates. This amount of noise will change two matching trips so much that our program will not merge them, since the differences are bigger than our tolerance levels.

On Figure 5.1 we can see a section of the *stops.txt* from the GTFS feed *sample-file.zip* before we added any noise to it. On Figure 5.2, we see the same section after adding a small amount of noise, namely to the values of *stop_name*. On Figure 5.3, we see the same section again, after adding a large amount of noise to the values of *stop_name* and *stop_lat, stop_lon*.

```
stop_id,stop_name,stop_lat,stop_lon
BEATTY_AIRPORT,Nye County Airport,36.868446,-116.784582
BULLFROG,Bullfrog,36.88108,-116.81797
STAGECOACH,Stagecoach Hotel & Casino,36.915682,-116.75167
NANAA,North Ave / N A Ave,36.914944,-116.761472
```

Listing 5.1: Section of *stops.txt* from sample-feed.zip before adding noise

```
stop_id,stop_name,stop_lat,stop_lon
BEATTY_AIRPORT,yN eoCnuytA rioptr,36.868446,-116.784582
BULLFROG,uBllrfgo,36.88108,-116.81797
STAGECOACH,tSgaceaohcH tole& C sani oD,36.915682,-116.751677
NANAA,oNtr hvA e / N AvA e,36.914944,-116.761472
```

Listing 5.2: Section of *stops.txt* from sample-feed.zip after adding noise to the values of *stop_name*

```
stop_id,stop_name,stop_lat,stop_lo
BEATTY_AIRPORT,yN eoCnuytA rioptr,36.870497335,-116.782659944
BULLFROG,uBllrfgo,36.881965733,-116.816864765
STAGECOACH,tSgaceaohcH tole& C sani oD,36.91588751,-116.75167
NANAA,oNtr hvA e / N AvA e,36.914944,-116.761472
```

Listing 5.3: Section of *stops.txt* from sample-feed.zip after adding noise to the values of *stop_name, stop_lat* and *stop_lon*

5.1.3 Evaluating two GTFS-Feeds

To evaluate our main gtfsMerge algorithm, we need to compare the merged feed to its original form. Our evaluation starts by reading in the two zip files using the same approach as the *gtfsMerge* program, giving us the dictionaries containing

the trip objects, stop objects, calendar objects etc. we need to compare. After that, we compare each line of the *trips.txt, stops.txt, stop_times.txt, calendar.txt, calendar_dates.txt, routes.txt* and *agencies.txt* files with each other. We check for each dictionary key if we can find the same key in the second feed. After that, we compare the lines by creating a string of the object, leaving out punctuation elements like whitespaces and commata, since they are irrelevant to the information of an csv-entry and may yet be different from one feed to the other. We then sort the strings and check for differences between the two. Comparing every line using this approach, we get the number of equal lines, different lines and additional lines, as well as a percentage of differences for two GTFS-Feeds. The lower our difference percentage is, the better our main algorithm has merged the split feed.

5.2 Results

In the following section, we take a look at the results obtained by using the algorithm on different GTFS-Feeds as well as a running time and storage analysis of the program. The running time has been obtained by using the program with minimal *print()* commands and without the memory profiler, in order to not let these aspects change the result.

The following results have been obtained using a personal laptop computer with a Intel Core i5-7200U CPU @ 2.50GHz, 8 GB of RAM and a Windows 10 64-bit operating system.

The percentages are the differences between the original feed and the merged feed. It is calculated by dividing the sum of the different and additional lines by the total number of lines in one feed:

$differences(\%) = \frac{differnt\_lines + additional\_lines}{total\_lines}$

Table 1 shows the ground truth we created by splitting *"sample-feed.zip"*. Each field gives the percentages of the differences between the original feed and the merged split feed, using different kind of comparisons and different levels of noise. The running time and memory usage are from the last merge operation, with *Mode 3* and *Noise*

2. As expected, we have small differences between the original feed and the merged feed when we are using *Mode 0* and *Mode 1*. Since not every case is covered, some trips that should be merged are left distinct. If we use the higher modes, every kind of case is covered and partially included trips are merged, giving us a percentage difference of 0% for *Noise 0* and *Noise 1*.

For *Noise 2*, we have high differences between the original feed and merged feed. This result is expected as well, since *Noise 2* creates large amounts of differences that are well above our margin of tolerance. The trips are no longer similar enough to be merged.

Table 2 shows the ground truth we by splitting an official feed *"chilliwack.zip"*. This feed has been premerged to allow for a more accurate evaluation. As with table 1, each field gives the percentages of the differences between the original feed and the merged split feed, using different kind of comparisons and different levels of noise. The running time and memory usage are from the last merge operation, with *Mode 3* and *Noise 2*.

Again, we can see a small amount of differences with Noise 0 and Noise 1 for Mode 0 and Mode 1. The differences become smaller as we increase the Mode. With Noise 2, the percentage is higher, again as expected, and decreases with Mode 2 and Mode 3.

We can see that we have an overall higher percentage of differences than in table 1. This is mostly due to the evaluation algorithm. Since we are dealing with an official real life feed, certain errors happened in the splitting and evaluation processes, like split trips being included in other trips, or split trips suddenly becoming equal to other split trips. These examples are rare and have been checked by hand to make sure the merge program has handled them correctly.

GTFS-Feed 1: sample-feed.zip

file	**lines**
trips.txt	11
stops.txt	9
stop_times.txt	28
calendar.txt	2
calendar_dates.txt	1
routes.txt	5
agency.txt	1

Mode	**0**	**1**	**2**	**3**
Noise 0	5.714%	5.714%	0.000%	0.000%
Noise 1	5.714%	5.714%	0.000%	0.000%
Noise 2	78.571%	78.571%	45.946%	45.946%

running time (milliseconds)	**38.86914**
memory usage (MiB)	**36.1797**

Table 1: Ground Truth 1 using "sample-feed.zip"

First table shows the number of lines for each text file in the GTFS feed.
Second table shows the difference between the original GTFS feed and the merged GTFS feed.
Mode specifies what kind of matching trips are handled: Mode 0 for equivalent trips only, Mode 1 for equivalent and included trips, Mode 2 for equivalent and partially included trips, Mode 3 for every kind of trip. *Noise* specifies the level of noise added to the feeds before merging: Noise 0 for no noise, Noise 1 for a small amount of noise, Noise 2 for a large amount of noise.
The third table shows the time it took to merge with *Mode 3* and *Noise 2* in milliseconds and the memory usage in MiB.

GTFS-Feed 2: chilliwack_premerged.zip

file	**lines**
trips.txt	462
stops.txt	291
stop_times.txt	11512
calendar.txt	4
calendar_dates.txt	1
routes.txt	10
agency.txt	1

Mode	**0**	**1**	**2**	**3**
Noise 0	2.414%	2.385%	0.488%	0.454%
Noise 1	2.539%	2.491%	0.548%	0.470%
Noise 2	37.766%	37.696%	16.088%	15.934%

running time (milliseconds)	**16633.26716**
memory usage (MiB)	**47.6719**

Table 2: Ground Truth 2 using "sample-feed.zip"

The First table shows the number of lines for each text file in the GTFS feed.
Second table shows the difference between the original GTFS feed and the merged GTFS feed.
Mode specifies what kind of matching trips are handled: Mode 0 for equivalent trips only, Mode 1 for equivalent and included trips, Mode 2 for equivalent and partially included trips, Mode 3 for every kind of trip. *Noise* specifies the level of noise added to the feeds before merging: Noise 0 for no noise, Noise 1 for a small amount of noise, Noise 2 for a large amount of noise.
The third table shows the time it took to merge with *Mode 3* and *Noise 2* in milliseconds and the memory usage in MiB.

Feed db_fv_premerged.zip

file	lines
trips.txt	6925
stops.txt	714
stop_times.txt	102100
calendar.txt	2557
calendar_dates.txt	1
routes.txt	1485
agency.txt	14
Size zipped:	**0.672 MB**
Size unzipped:	**4,28 MB**

Feed ch_fv.zip

file	lines
trips.txt	17625
stops.txt	5897
stop_times.txt	161471
calendar.txt	12455
calendar_dates.txt	12346
routes.txt	1544
agency.txt	62
Size zipped:	**1,21 MB**
Size unzipped:	**8,28 MB**

Merged Feed: "ch_fv + db_fv_premerged.zip"

file	**lines**
trips.txt	11448
stops.txt	1862
stop_times.txt	142879
calendar.txt	3454
calendar_dates.txt	1178
routes.txt	2722
agency.txt	76

running time (milliseconds)	**38'640'449.180**
running time (minutes)	**644.007**
memory usage (MiB)	**4445.705**
Size zipped:	**1.7 MB**
Size unzipped:	**6.43 MB**

Table 3: Merge result 1 using feeds "ch_fv.zip" and "db_fv_premerged.zip"

First two tables show the number of lines for each feed as well as the size of the feed
The second table shows the number of lines for each file in the merged feed as well as the time it took to merge the feeds,the memory usage and the size of the merged feed

Feed fraser_valley_feed.zip

file	lines
trips.txt	483
stops.txt	243
stop_times.txt	10461
calendar.txt	5
calendar_dates.txt	0
routes.txt	15
agency.txt	1
Size zipped:	**0.118 MB**
Size unzipped:	**0.845 MB**

Feed comox_valley_feed.zip

file	lines
trips.txt	2927
stops.txt	638
stop_times.txt	102075
calendar.txt	7
calendar_dates.txt	2
routes.txt	23
agency.txt	1
Size zipped:	**0.872 MB**
Size unzipped:	**6,03 MB**

Merged Feed: "commox_valley_feed + fraser_valley_feed.zip"

file	lines
trips.txt	1576
stops.txt	878
stop_times.txt	51350
calendar.txt	12
calendar_dates.txt	1
routes.txt	38
agency.txt	2

running time (milliseconds)	**2'555'623.13175**
running time (minutes)	**42.59**
memory usage (MiB)	**285.4**
Size zipped:	**0.393 MB**
Size unzipped:	**3.54 MB**

Table 4: Merge result 2 using feeds "commox_valley_feed.zip" and "fraser_valley_feed.zip"

First two tables show the number of lines for each feed as well as the size of the feed
The second table shows the number of lines for each file in the merged feed as well as the time it took to merge the feeds,the memory usage and the size of the merged feed.

5.3 Conclusion

The goal of this program is to merge overlapping GTFS feeds in a way that the output is a valid GTFS feed retaining all of the relevant information while merging and deleting equivalent, redundant and overlapping trips. The data handling has been achieved using a csv reader and a 2D grid of lists to map the trips using their geographical location. The matching has been achieved by using datetime objects and the haversine formula to compare stop times and stops. our output is a valid merged GTFS feed.

6 Future Work

In this section, we will take a look at possible improvements to the program that could be done in future projects.

6.1 Managing the shape_id for merged trips

GTFS feeds have an optional text file called *shapes.txt*. In this file, the entries describe the physical paths that a vehicle takes, consisting of a sequence of points. Each trip can contain a *shape_id*, matching it to a shape in shapes.txt.
In our program, we ignore these text files and entries. If we merge two trips and create a new trip, the *shape_id* needs to change, since it no longer fits the shape it references. Since we merged two trips, the referenced shapes need to be merged too.
Right now, we avoid this problem by setting the *shape_id* of the new trip to "". This way, the new trip will not reference a incorrect shape from *shapes.txt*. This is a aspect of our program that could be improved my creating a function that automatically merges two shape objects if two trips are merged.

6.2 Managing entries that do not belong to any trip

When reading in our GTFS feed and building the program objects, we fill the trip object with the other *stop-, stop_times-, route-objects* ect. We only fill them with the objects that reference that specific trip. This means that entries that do not match any trip in the feed are ultimately lost.

A useful addition to our program would be a function that takes care of "orphaned" objects and adds them to our output. These objects could also be merged again to check for duplicates that are already assigned to a trip object, to increase the accuracy.

6.3 Using optional attributes to put out more data

At the current state of our program, we only manage the required text files from the two GTFS feeds. We also only return the required text files in our merged GTFS feed. A useful addition to our program would be a function that would handle the optional text files by taking into consideration which kind of optional text files and optional attributes are in the original feeds. This would increase the completeness of our output.

6.4 Improving the matching of trips using different attributes

At the current state of the program, we mainly use the attributes *arrival_time, stop_lat* and *stop_lon*. If we wanted to increase running time, we can start looking for additional arguments that would give us more information on which trips are to be merged and which are discarded. With additional attributes, it could be possible to shrink the number of trips to be compared with each other even further.
One way would be to implement a sort of data base or search engine that would return the closest match for a given stop name. This way, we would not need to rely on the haversine formula to compare the stop objects and running time and accuracy would increase.

6.5 Managing Overlapping Service Dates

Our current program compares calendar dates by checking if the start- and end dates of two trips are the same, or if one is included in the other. However, if we have overlapping dates, the trips are handled as if they have different service times. To improve accuracy in matching trips, we could check if two trips have overlapping dates and merge them, taking the earlier start date and the later end date as the new calendar dates for the trip.

6.6 Improving Memory Usage

Our current algorithm is using hash keys and sets to avoid comparing two trips more than once. For big data sets, these sets of hash keys can become very memory consuming. To avoid problems for machines with less memory, it would be advisable to employ a more efficient way to store the data.
One way to achieve this would be to use data banks in combination with SQL to store information of bigger feeds efficiently.

6.7 Running time optimization

In programming, running time is always an important area to work on in order to improve a program. In our case, we do not need to compute and put out results in real time, so a low running time is not of the highest priority. However, certain improvements can still be made to decrease running time and increase the usefulness of the program.
The function that takes the longest time to compute is *compareTrips()*. In this function, we have certain nested loops that are running in $O(n^2)$ in the worst case, which is not an optimal complexity. Improving this function can be done by only comparing trips that are similar in the first place. This means only comparing trips where we can determine a certain similarity beforehand, and discarding the rest of

the trips to be compared differently. This is already done by our 2D grid, but could be extended even further.

One of these similarities could be the service dates. By using the different kind of service dates to add another dimension to our 2D grid, we could reduce the numbers of trips to compare to each other. This would improve our running time, although we can't be sure how by how much, since it is probable that the majority of trips will frequent on the same service days from Monday to Friday.

7 Acknowledgments

First and foremost, I would like to thank Prof. Dr. Hannah Bast for enabling me to work on this project.
I would also like to thank Patrick Brosi for his supervision, creative input and support.
I would also like to thank my friends and family for supporting me throughout my studies.

Bibliography

[1] Google, “Merge.” https://github.com/google/transitfeed/wiki/Merge.

[2] P. Tran, “Semantischer vergleich von fahrplandaten,” *Bachelorarbeit*, 2017.

[3] Z. Zhang, “Public-transit data extraction from openstreetmap data,” *Bachelorarbeit*, 2017.

[4] “General transit feed specification.” https://developers.google.com/transit/gtfs/.

[5] N. A. Rooy, “Calculate the distance between two gps points with python (vincenty’s inverse formula).” https://nathanrooy.github.io/posts/2016-12-18/vincenty-formula-with-python/.

[6] “Timsort.” https://svn.python.org/projects/python/trunk/Objects/listsort.txt/.

[7] “Csv file reading and writing.” https://docs.python.org/3.7/library/csv.html.

[8] B. Rouberol, “Haversine.” https://github.com/mapado/haversine.

[9] “datetime–basic date and time types.” https://docs.python.org/3/library/datetime.html.

[10] “Memory profiler.” https://pypi.org/project/memory-profiler/.

[11] “Textiowrapper.” https://docs.python.org/3/library/io.html.

[12] “shutil.” https://docs.python.org/3/library/shutil.html.