

Master's Thesis

---

# Automatic Generation of Frequency Maps for Public Transit Networks

---

Bhashitha Gamage

Examiner: Prof. Dr. Hannah Bast

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

May 11<sup>th</sup>, 2018

**Primary Reviewer**

Prof. Dr. Hannah Bast

**Secondary Reviewer**

Prof. Dr. Georg Lausen

**Supervisors**

Prof. Dr. Hannah Bast

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

This thesis addresses the problem of automatically generating the frequency map of a public transit network when given its GTFS [1] data feed. Normally, the drawing of a frequency graph is completely done manually and doing it automatically is extremely complicated. Our approach consists of a few major steps including extraction of the transit graph from the GTFS data, implementation of the frequency finding algorithm, extraction of the frequencies among interested nodes, creation of the frequency lines and finally the drawing of frequency lines and nodes in a convenient way to resemble a schematic map. The last step which is the drawing of a frequency map automatically was the most challenging step, and several approaches experimented until satisfactory results were achieved. We explain each of this steps in detail and discuss how the appeared challenges are solved and overcome in a scientific manner. We tested our approach against several examples GTFS Feeds, evaluated it concerning the running time and the frequency coverage and compared our results to the official, hand-drawn frequency graph published by the Swiss Federal Railways.

# Zusammenfassung

Diese Arbeit befasst sich mit dem Problem der automatischen Generierung von Taktfrequenz-Karten aus Fahrplandaten in Netzen des öffentlichen Verkehrs. Derartige Karten wurden bisher vollständig von Hand gezeichnet, da eine automatische Generierung sich als enorm schwierig erwiesen hat. Wir beschreiben in dieser Arbeit folgende Schritte: die Extraktion eines "Transit-Graphen" aus GTFS-Daten [1], die Implementierung eines Algorithmus zum Finden der Taktfrequenzen, die Extraktion und Filterung dieser Frequenzen, die Erstellung von "Frequenz-Linien" auf dem Transit-Graphen und schließlich das Zeichnen dieses Graphens in einer Weise, die einer schematischen Netzkarte ähnelt. Der letzte Schritt - das schematische Zeichnen dieses Graphens - war am herausforderndsten. Wir haben mehrere Ansätze evaluiert bis zufriedenstellende Ergebnisse erreicht waren. Jeden der genannten Schritte erklären wir im Detail und diskutieren unsere Lösungsansätze. Unser Ansatz wurde auf verschiedenen GTFS-Feeds getestet, bezüglich Laufzeit und Frequenz-Abdeckung evaluiert sowie mit einer offiziellen, hand-gezeichneten Taktfrequenzkarte der Schweizerischen Bundesbahnen verglichen.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Preliminaries</b>	<b>5</b>
2.1	General Transit Feed Specification . . . . .	5
2.2	Geographic Information System (GIS) . . . . .	6
2.3	GeoJSON . . . . .	7
2.4	QGIS . . . . .	8
2.5	Dot language and Graphviz . . . . .	9
2.6	Frequency Graphs . . . . .	10
<b>3</b>	<b>Related work</b>	<b>12</b>
<b>4</b>	<b>Implementation</b>	<b>15</b>
4.1	Extraction of transit graph from GTFS data . . . . .	16
4.2	Frequency finding algorithm. . . . .	25
4.2.1	Implementation of Frequency Finding Algorithm . . . . .	25
4.2.2	Human-friendly frequency finding and filtering frequencies . . . . .	35
4.3	Extraction of the frequencies among interesting nodes and creation of frequency lines. . . . .	37
4.4	Drawing of frequency lines and nodes in a way that resembles a schematic map	49
4.4.1	First approach by creating a web application using Leaflet . . . . .	49
4.4.2	Second approach using GeoJSON and QGIS . . . . .	52
4.4.3	Third approach using GeoJSON and Octi . . . . .	57
4.4.4	Fourth approach using Graphviz . . . . .	59
4.4.4.1	Neato layout engine . . . . .	63
4.4.4.2	Prevent overlapping of parallel edges in Frequency Map . . .	68
4.4.4.3	Merging the unidirectional frequency lines and creation of the bidirectional frequency lines . . . . .	77
<b>5</b>	<b>Evaluation</b>	<b>87</b>
5.1	Evaluation of the time required to find frequency coverages using the frequency finding algorithm and the human readable frequency finding algorithm . . . .	87

5.2	Evaluation for the time required to build the transit graph . . . . .	88
5.3	Evaluation of the frequency graphs that were rendered on the Swiss Railway Network GTFS dataset and the Deutsche Bahn GTFS dataset . . . . .	90
5.4	Evaluation on the trips covered by the frequency coverages out of the total trips	92
5.5	Evaluation between the automatically generated frequency graph and the manually created Switzerland Timetable-2017 graph . . . . .	94
<b>6</b>	<b>Future Work</b>	<b>97</b>
<b>7</b>	<b>Acknowledgments</b>	<b>101</b>
	<b>Bibliography</b>	<b>101</b>

# List of Figures

1	Frequency graph between Node1, Node2 Node 3 . . . . .	10
2	Frequency graph between Node1, Node2 Node 3 with geographic location . .	11
3	Metro map generated by metro map drawing tool on Karlsruhe tram system .	13
4	Google GTFS reference page [2] . . . . .	17
5	trips.txt [3] . . . . .	18
6	Class diagram of Trip class . . . . .	18
7	Class diagram of GTFSColoumnParser class . . . . .	19
8	routes.txt [3] . . . . .	20
9	Class diagram of Route class . . . . .	20
10	stops.txt [3] . . . . .	20
11	Class diagram of Stop class . . . . .	21
12	calender.txt [3] . . . . .	21
13	Class diagram of Service class . . . . .	22
14	Class diagram of Range class . . . . .	22
15	calendar_dates.txt [3] . . . . .	22
16	stop_times.txt [3] . . . . .	23
17	Class diagram of Node class . . . . .	23
18	Class diagram of Edge class . . . . .	24
19	Class diagram of Time class . . . . .	24
20	Transit graph built by reading GTFS dataset . . . . .	25
21	Departure times between Node A and B . . . . .	26
22	Class diagram of GapRecord class . . . . .	30
23	Set of identified frequency covers between Basel Bankverein - Basel Aeschenplatz 1	33
24	Set of identified frequency covers between Basel Bankverein - Basel Aeschenplatz 2	34
25	Class diagram of Boundary class . . . . .	36
26	Example of nodes and edges overlapping and edges draw as straight lines . . .	38
27	The diagram with two consecutive nodes named A and B and out connections from Node A to Node B . . . . .	39
28	The diagram with multiple nodes and out connections. . . . .	40



29	Class diagram of FrequencyLine class . . . . .	45
30	Class diagram of TripAndDuration class . . . . .	45
31	An image of Swiss railway network rendered by the Leaflet . . . . .	51
32	Displaying the identified frequencies between Walterswil-Striegel and Kün- goldingen . . . . .	52
33	The initial version of the Switzerland railway map rendered by QGIS . . . . .	55
34	Class diagram of LotLonToWebMercator class . . . . .	55
35	The Switzerland railway map rendered by QGIS with web mercator coordinates	57
36	Subway network of Vienna rendered by Octi . . . . .	58
37	Graph generated by neato layout engine using the orthogonal edge style and fixed node positions . . . . .	64
38	Rendered frequency graph using Graphviz Neato layout engine . . . . .	66
39	Overlapping of parallel edges between Luzern and Zug . . . . .	67
40	Overlapping of parallel edges between Zuürich HB and Pfäffikon SZ . . . . .	68
41	Graph rendered after introducing dummy nodes . . . . .	70
42	Maze consisting with cells . . . . .	72
43	The diagram with <code>n_dad</code> between snodes . . . . .	73
44	Graph generated with the new <code>updateWt()</code> function and the coloured frequency lines . . . . .	76
45	Parallel edges between the Luzern and the Zug are not overlapping . . . . .	77
46	Frequency graph that generated with the bidirectional frequency lines . . . . .	84
47	Bidirectional edges between Zürich HB and Winterthur . . . . .	85
48	Bidirectional edges between few nodes . . . . .	85
49	Frequency graph generated on Deutsche Bahn GTFS data . . . . .	90
50	Frequency graph generated on Swiss railway GTFS data . . . . .	91
51	Diagram of Switzerland Timetable-2017 Frequency Graph [4] . . . . .	94
52	Illustration of connections as well as splitting/ joining of two trains [4] . . . . .	96
53	Diagram of node A,B, C and their frequency covers . . . . .	98
54	The diagram with node A, B, C and their frequency covers. In connections of node B, connected with its out connections . . . . .	98
55	Format of displaying 15 minute, 20 minute, 30 minute frequencies in Switzerland Timetable-2017 graph [4] . . . . .	99
56	Edges with different line thickness and colour combinations . . . . .	99

# List of Tables

1	Two sample graphs described in Dot language . . . . .	59
2	Two sample graphs rendered by Graphviz . . . . .	60
3	This table shows different ways of connecting orthogonal edges with node and the angle between the node and the edges . . . . .	82
4	Running time of frequency finding algorithm . . . . .	88
5	Running time of human-friendly frequency finding algorithm . . . . .	88
6	For the selected nodes, the total number of departure times covered by frequency coverages in the Swiss GTFS dataset . . . . .	92
7	For the selected nodes, the total numbers of departure times covered by the frequency coverages in the Deutsche Bahn GTFS data . . . . .	93

# List of Algorithms

1	Algorithm for decide initial value for K,pseudo code : . . . . .	29
2	Algorithm for frequency finding is as follows. . . . .	30
3	Algorithm for finding the arithmetic progressions starts with the xth item of the records collection. . . . .	31
4	Algorithm for check availability of departure times . . . . .	36
5	Algorithm to find the frequency between two distantly located nodes . . . . .	41
6	Algorithm for initiating of the frequency line generation: Part 01 . . . . .	42
7	Algorithm for initiating frequency line generation Part 02 . . . . .	43
8	Algorithm for frequency line generation: Part 01 . . . . .	47
9	Algorithm for frequency line generation: Part 02 . . . . .	48
10	Algorithm for convert the latitude longitude values to web mercator projection coordinates. . . . .	56
11	Algorithm for increase the space between parallel edges . . . . .	74
12	Algorithm for introducing colors for the frequency lines . . . . .	75
13	Algorithm for frequency lines checking and processing . . . . .	78
14	Algorithm for merging unidirectional frequency lines: Part 01 . . . . .	80
15	Algorithm for merging unidirectional frequency lines: Part 02 . . . . .	81

# 1 Introduction

Public transportation is a crucial part of a country and it affects a nations' energy, economy and environments- all of which influence the quality of life of the people living in the country. The rate of people using public transportation is increasing every day and local communities are expanding public transit services. Every part of the society benefits from the public transportation. For example individuals, families, communities and businesses. The main reasons for this are the following:

- Public transportation enhances the opportunities for people.
- It is a solution for traffic congestion.
- It drives communities towards economic growth.
- It is cost effective.
- It reduces the gasoline consumption.
- It reduces the carbon footprint.

Because of the reasons mentioned above, government authorities and communities are going to increase and enhance the uses of public transportation.

Due to the advancement of the information and communication technologies, transit agencies could collect and generate transit data at a high frequency. Additionally, there is an increasing demand for the visualization of transit data which will lead to effective planning, operational investments and the enhancement of the transit performance. This transit data can be used to improve the public transportation for the public community.

In the last century, displaying public transit data was very expensive because of the cost of the data collection. The data collection was done primarily via manual labour. However, the advancement of digital technologies such as Global Positioning System, Automatic Passenger Counters, Automatic Vehicle Location and Google Transit Feed Specification have reduced the difficulties in data gathering and data representation.

Similar to the data gathering, the visualization and representation of public transit data are

very important for the commuters who use the transport network. Nowadays these data are represented in various ways including metro maps, mobile apps, web pages etc. One of the main ways of representing these public transit data is via the use of frequency maps.

The most essential and basic use case of the frequency maps is not about the quality of the service, rather it is about the availability of the service, which is the existence of transportation service when a user needs it. This is the primary question which is addressed by frequency maps.

Normally, transit services often emphasize “where” the services exist, but not when it exists. However, the existence of “where” and “when” both are equally important when deciding the existence of the service. For urban transits, the frequency is an important factor when determining the usefulness of a service. The most crucial factor in deciding a transit service is the waiting time. If the waiting time increases, it will decrease the usefulness of the transit service to the user. So in summary, the waiting time will depend on the frequency of the transit service. It is also important to note that the frequency will determine whether connections are easy to make or not. That means the frequency will decide, whether or not transit is a well functioning network.

Most transit agencies services can be categorized into three main parts.

- Frequency Network.

This transportation service is available throughout the day, typically within every 15 min or less

- Infrequent All-day services.

This means that the transit network often relies on timed connection throughout the day.

- Peak-only service

This exists only during the peak hours. This contains long commuter-express routes. These type of services add much complexity to a frequency map, but they include very specialized services.

E.g.: A Frequency map consists of the frequencies of the public transport medium which is available during working hours or/and during the entire day. An example of a frequency could be every 20th minute of each hour during the normal working hours, a train is leaving from station A and arriving at station B at the 50th minute of each hour etc.

The importance of representing frequencies in a map is very important for making a decision.

Frequency variance has been represented using different types of lines. The main purpose of this is to make the frequency network stand out from rest of the map.

These frequency maps facilitate the commuters who use the public transport medium by giving them the available departure time from a starting station and the arrival times at the destination station. Additionally, frequency maps can give the information on how frequent (once per every hour, half an hour, once per every two hours etc.) the transport medium is travelling between station A and B and, what category the transport medium is. When we consider a railway network the category lets the users know, whether it is an inter-city express, regional express or S-Bahn etc. Cities can be restructured in accordance with the importance of the transits regarding the citizen's needs. People who value transits will live near transits. This is one of the main advantages that a frequency map has over the normal map. Frequency maps are very useful for people making decisions when deciding about different things: where to live, where to shop, where to buy a property/land, and also where to start a business. Such decisions can be made/influenced via the aid of a frequency map. For example, a family who has children will likely settle near a transit service, based off of information from a frequency map. Another possible example would be the location of an office according to the availability of peak hours only services which will help with the transportation of the employees.

In summary, frequency maps are a huge part of the decision making for commuters. It is important to differentiate the frequency of transits in a map because this will both, directly and indirectly, help its' primary and secondary audience in their decision making. This directly affects the economy of the community, quality of life and the growth of the economy.

Normally, the drawing of a frequency map is entirely done manually with the assistance of graphing tools. The automatic drawing of an end to end frequency map which is ready to print is currently somewhat tedious. The reason for this being that a human is always needed in order to minimize the number of edge crossings, align parallel lines, and drawing lines that are infrequent (or frequencies with exceptions). Even the automation of some of the steps of the drawing of a frequency map is extremely difficult. This can directly impact the unexpected changes in schedules because of this time-consuming process. It should also be noted that in some scenarios, it could be hard to distribute such changes to the commuters in a live-time manner. The reduction of the manual labouring will decrease the associated costs.

Because of the significance of the frequency maps, it is a must to have that an automated process to develop maps according to the requirement.

By using aforementioned scenario as the main demonstrative, the objective of this thesis is to automatically generate frequency maps with high efficiency and accuracy which can then be used by cartographers as a blueprint when drawing more artistic frequency map manually.

The mechanism presented in this thesis will make the drawing of frequency maps easy and will allow the generation of blueprints for frequency maps on a daily basis - if required.

This thesis is organized as follows. Chapter 2 consists of fundamentals about GTFS data and basic concepts of graphs and graph drawing. Chapter 3 discusses the related work that is done in this field and the theories which are adapted during the implementation. Chapter 4 includes the explanation of steps that are used to generate a frequency map. The first step is the extraction of the transit graph from the GTFS data. The second step is to extract the frequency from this transition graph. The third step is to clean up these extracted frequencies, by validating their journey duration. The fourth and final step is the hardest, in which the data is drawn in a nice fashion which resembles a schematic map. Graph drawing itself is a domain where much research is being conducted. Instead of implementing a graph drawing tool from scratch, the main idea was to evaluate existing open source graphing tools/ libraries and to select the best matching tool and then modify it as per needs and use them to render frequency graphs. Chapter 05 includes the evaluation of the process of automatically drawing frequency maps. Variations of the running times of different versions of frequency finding algorithms, processing power and memory required will be covered here. Next, the tool that is implemented will be tested on two GTFS data sets and will draw frequency maps for these data and then compare them. Here it also compares the results of the graphing tool (Octi) developed under the chair of Algorithms and Data Structures of University of Freiburg [5] against to results of the Graphviz tool. Chapter 06 is dedicated to the future works. Under the future works section, it discusses further improvements and modifications that could be done in order to fine-tune the results. Finally, it discusses the barriers that need to be overcome to overcome to render state of the art frequency graphs.

## 2 Preliminaries

The main objective of this chapter is to describe the main theories, concepts and practical scenarios used in the thesis. This includes the General Transit Feed Specification (GTFS), Geographic Information System, Frequency Graphs and some other technologies which are adapted during the implementation.

### 2.1 General Transit Feed Specification

GTFS [1] was first introduced as a common format for public transportation schedules and associated geographic information by TriMet in Portland, Oregon. It was one of the first public agencies who tried to tackle the problem of online transit trip planners with Google Inc. TriMet worked with Google to identify a format to transfer their transit data into Google Maps [6]. This format was then called General Transit Feed Specification which then released as Google Transit. After releasing it in 2005, GTFS has been the most popular data format which is used to describe fixed-route transit services in/around the world. Currently, approximately 5900 agencies are included in the Google Transit coverage. Since GTFS data is publicly available, many 3rd parties use these data for trip planning, maps, mobile data, visualization, accessibility, analysis tools for planning, timetable creation, and real-time information systems [7]. By considering these vast possibilities, the objective of this thesis came as a module of frequency based data modelling. Public transportation data handling is an area which is widely researched. However, due to the huge timetable data which needs to be handled in the process, it is a challenging area to research. Because of millions of departures happens for a day, the timetable data compression is a crucial area for public transportation data handling.

A General Transit Feed Specification is a collection of series of text files. This is a combination of transit information such as stops, routes, trips and other schedule data. Developers could use those published GTFS data for their applications. Recently interest has been growing to create new types of services based on same transit data. Currently, route planning in large public transportation networks is operated at periodic time intervals. Each of the time events is stored separately. When a set of connections has sufficient periodicity, it becomes more efficient to store time range (start time and end time) and frequency.



For this thesis, the main purpose of GTFS data is to process and visualize transit data in a readable manner for the users. The main limitation of consuming GTFS data directly or from a stored database was that it does not have good accessibility to the frequency between transits. The algorithm mentioned in Frequency-Based Search for Public Transit [8] research paper was adapted in this thesis to find the frequency coverage, and it was adjusted and improved according to the requirement.

GTFS is a series of 13 unique text files which has been compressed into a .zip file. Each text file is formatted into a comma separated value file. For each text file, it contains a header field and description of header fields, which are prescribed according to the specification. Though there are 13 text files, these are related to one another by sharing one or more common fields in both text files. This behaviour is similar to the relational database.

The most important feature in GTFS feed is the proper use of required fields and files, and the proper relationship among the data contain in those files. Because of this consistency, it allows many consumer-based applications to use GTFS data. The transit agency must provide the data with internal consistency. For example, fields which have been shared among different text files must have identical names. To validate GTFS data, Google provides the Google Feed Validator [9] which is an open source tool.

With the help of GTFS data, Open Data and Open Governance concept can be established within communities. In this broader context, data can be used for personal use and to monitor and evaluate provided services. For example, citizens can compare the actual experience with available transit data to get a conclusion on the quality of service. Most of the governments' have directly been involved in quality of the data provided to the public by agencies and the accessibility of data to every citizen which empowers citizens with valuable data.

There are some major issues which need to be handled when going forward with GTFS data. Such as resolving and collecting data from complex institutional settings, uses of "informal services", regular updating of the data feed, consistency of data and the use of real-time data.

## **2.2 Geographic Information System (GIS)**

GIS is a framework for gathering, managing and analyzing data. GIS is combined with the science of geography. It integrates many types of data. The main objective is to analyze spatial location and organize layers of information to visualize using maps and 3D scenes. In short, the main role of GIS is to use an input set of raw data and convert it into useful output information. The other primary function of GIS is the ability to query the database by using a selection of attributes or locations to find the relationship between different results.

GIS divides the world into two main parts objects and attributes which can then be represented spatially by raster or vector datasets.

**Raster Datasets:** Datasets which comes from grids e.g: aerial photos.

**Vector Datasets:** Datasets which comes from mathematical calculation and functions. E.g: points, lines, polygons.

The development of the spatially oriented database, navigation services, web map services and localization of data has allowed for the public to benefit from it. Google Maps is the most used GIS application in the world at the moment. Most of the local governments use GIS for things such as planning applications, parking suspensions, highway maintenance, transport, environmental aspects, emergency planning and for street furniture. Nowadays, both public and private sectors use GIS information on day to day activities.

GIS provides many benefits for its users, such as cost saving results with efficiency, assistance in decision making, improve communication, - it is a good database for storing of geographic information records.

## 2.3 GeoJSON

GeoJSON is a Geographical Information Format which represents geographic features, data in an open standard format using JSON [10]. It differs from GIS because it was implemented by developers who worked with the internet. Regarding the format of GeoJSON, it was started on 2007 and finalized on 2008. There are many mapping packages which support GeoJSON. GeoJSON supports different geometry types, such as Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon. Geometric objects which contain additional properties are named as Feature objects. A set of Features can be represented by FeatureCollection objects.

```
1 {  
2   "type": "Feature",  
3   "geometry": {  
4     "type": "Point",  
5     "coordinates": [125.6, 10.1]  
6   },  
7   "properties": {  
8     "name": "Dinagat Islands"  
9   }  
10 }
```

### **Example of a Sample GeoJSON. [11]**

**Coordinate:** This is a single number representing a single dimension which can be further divided into longitude and latitude. The format of coordinates in GeoJSON is as same as numbers in JSON which contains a single decimal format.

**Position:** This means an array of coordinates in an order. A point represents a place on earth. In GeoJSON order of coordinates need to be [longitude, latitude, elevation].

**Geometry:** These are shapes. All simple geometries in GeoJSON include a type and collection of coordinates. eg: Points, LineStrings, Polygons

**Features:** Combination of geometry and positions

Because of GeoJSON, different data providers provide geo data in the same format so that developers can consume data regardless the API provider. Clients will be independent of the backend. Due to available API, data have the same data format. Geo data are a regular javascript object which is easy to consume.

## **2.4 QGIS**

QGIS [12] is an open source GIS software which allows editing, viewing and the analysis of geospatial data. QGIS supports both raster and vector layers. Vector data can be stored either points, lines or polygons. Multilayers of raster images are supported. The software can be used to georeference images.

As input types, the user can use shapefiles, coverages, personal geo databases, PostGIS, MapInfo and many more types. QGIS is combined with other open source GIS Packages such as GRASS GIS, PostGIS and MapServer. It also supports PostgreSQL/PostGIS, SpatiaLite and MYSQL databases.

QGIS is high in interoperability, and it provides around 148 base maps, high geoprocessing availability, contains semi-automatic classification plugin, contains project level/layer level/-package level variables, the package includes high data visualization/styling methods, and contains virtual layers and geometric layers.

## 2.5 Dot language and Graphviz

Dot [13] is a language that is used to explain graphs which are directed as well as undirected. More about the dot language can be found on its wiki page [14], and some of this information will be included here for the reason of self-contained purpose of the thesis. Once a graph is explained in dot language, that file should be saved with dot or gv extension, and then various programs can be used to process that dot file and to draw the graph. Some of these programs are

- dot
- neato
- twopi
- circo
- fdp
- sfdp etc.

The purpose of these programs as well as their output graphs, will be further explained in the implementation chapter. The tool that we used to process dot language files is Graphviz [15]. More information about Graphviz also can be found in its web page [16]. Graphviz supports most of these dot rendering programs, and it should be installed in a host computer before using it. Then using the following command, we can use any of the programs listed above to draw a graph using the dot language file.

```
dot -Kneato -Tsvg GraphInDot.dot -o Graph.svg
```

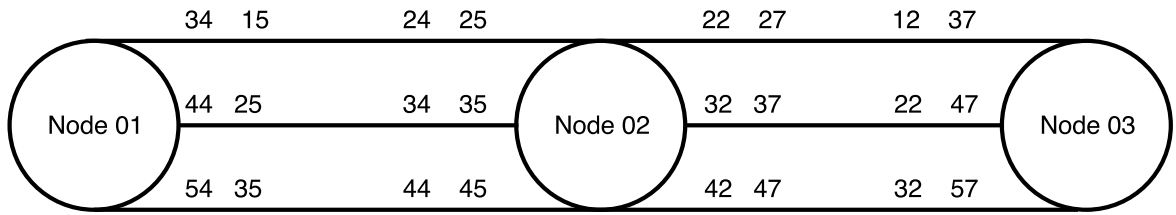
- -K: Specifies which default layout algorithm to use, overriding the default from the command name. For example, running dot -Kneato is equivalent to running neato.
- -T: Set output language to one of the supported formats [17]. By default, attributed [17] dot is produced.
- GraphInDot.dot : This is the dot language file which describes the graph in dot language.
- -o: Write output to file outfile. By default, output goes to stdout.

Graphviz supports many more commands rather than above mentioned basic commands and detailed use of these commands can be found in its command line invention [18] page.

## 2.6 Frequency Graphs

Frequency graphs are the graphs which give information about the frequency coverage in a public transit network. Revealing the frequency coverages in a public transit network helps the commuters who use the transit network on a daily basis. It is generally accepted that well-designed frequency graphs can better serve the commuters as opposed to ordinary timetables, so long as that transit network consists of good frequency covers.

At this point, it is more convenient to explore some basic principles about the frequency graphs and how to read the frequency graphs properly. A frequency graph consists of nodes and edges. These edges are the frequency lines that represents the frequency coverages between nodes. The following is a sample diagram of a frequency graph.



**Figure 1:** Frequency graph between Node1, Node2 Node 3

The diagram mentioned above is a simplified version of a frequency graph. Nodes are represented by circle shapes, and the frequency coverage is represented by drawing frequency lines between the nodes.

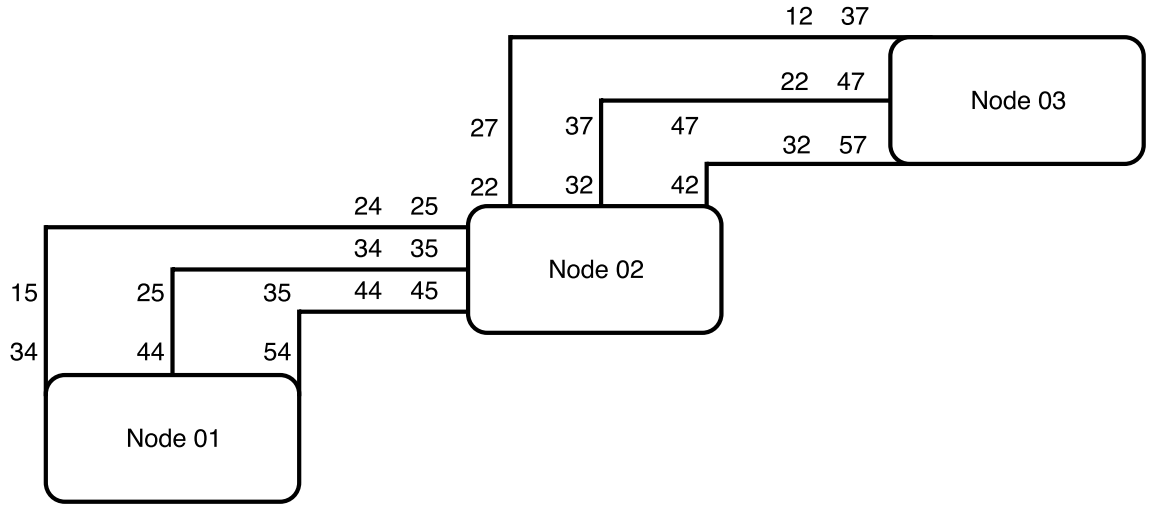
According to the above frequency graph the following information about journeys between nodes can be extracted.

- Every hour at the 15th, 25th and the 35th minute a journey starts from Node 01 and these journeys arrived at Node 02 at the 25th, 35th and 45th minute respectively.
- Every hour at the 27th, 37th and the 47th minute a journey starts from Node 2 and these journeys arrived at Node 03 at the 37th, 47th and 57th minute respectively.
- Every hour at the 12th, 22th and the 52th minute a journey starts from Node 3 and these journeys arrived at Node 2 at the 22th, 32th and 42th minute respectively.
- Every hour at the 24th, 34th and the 44th minute a journey starts from Node 2 and these journeys arrived at Node 1 at the 24th, 44th and 54th minute respectively.

Normally frequency graphs do not contain frequency covers for the entire day because during

the work hours (ordinarily 7:30 in the morning until 17:30 in the evening) there are more connections between nodes in comparison with the early hours or later hours in the day. Therefore, frequency graphs are designed in a way that most commuters can benefit from. This is achieved by drawing the frequency graph which covers at least official working hours on a normal weekday. Even though the departure minute and arrival minute are mentioned, the travel duration is not shown here. This could be included along with the edge, as a separate label. However, surprisingly most of the frequency graphs which already exists out there has no mention of the travel time. This might be due to the reason that the most of the commuters who use the public transit network are aware of the average travel time which it takes to travel between nodes.

In the diagram above, even though all the frequency covers are mentioned, that diagram does not give any information about the geographical location of the nodes and their relative location to each other.



**Figure 2:** Frequency graph between Node1, Node2 Node 3 with geographic location

In the above frequency graph, it also provides the notion of information about the geographical location of the nodes. The nodes are not aligned in a straight line, and therefore the frequency lines between nodes cannot be straight lines, because that will reduce the quality of the graph and the readability of this graph. Therefore orthogonal edges are introduced and routed in a way that minimizes the unnecessary edge crossings. This graph is more convenient for the commuters when compared to the previous frequency graph. However, the drawing of a frequency graph like this is not trivial and easy. The implementation chapter is dedicated for this purpose, and there it explains how to render a frequency graph like this automatically, with no/less human intervention.

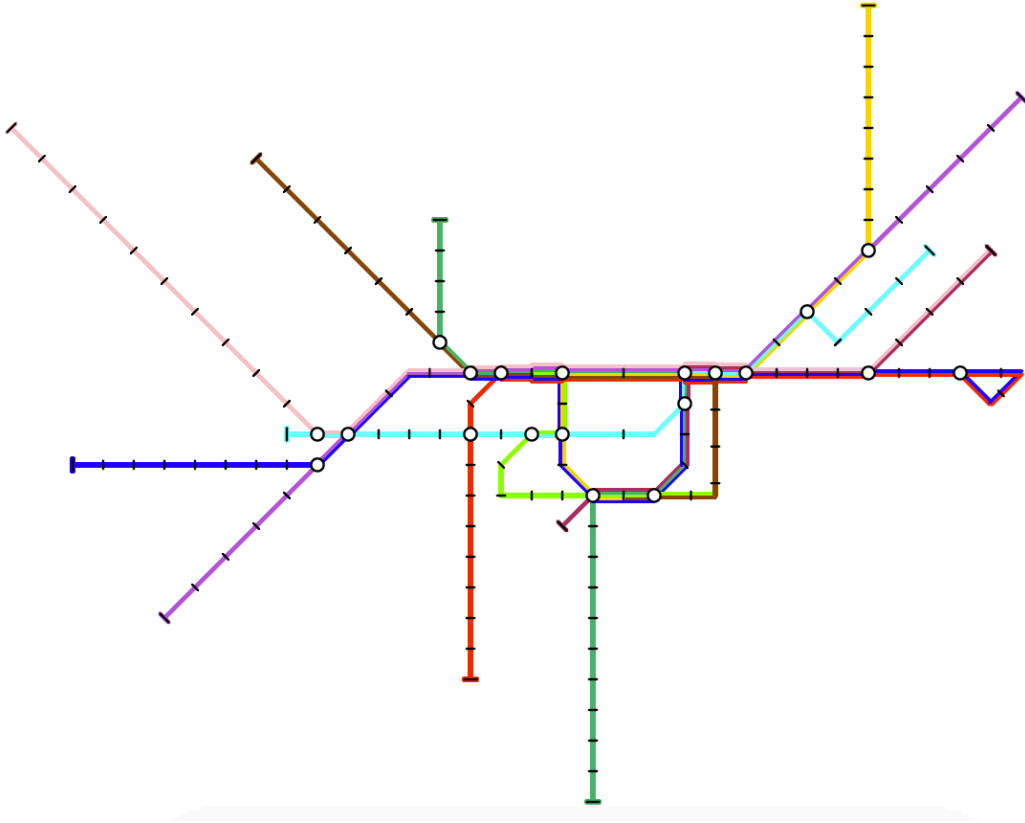
### 3 Related work

This chapter is dedicated to discussing the work that has been done regarding the automatic generation of the frequency maps using GTFS data. A frequency map consists of frequencies of the transits. Therefore the mechanism of the finding of the frequency coverages among a set of departure times is a crucial factor to generate a frequency graph. Finding of the arithmetic progressions given a number sequence which is in ascending order is well discussed and approached in Frequency-Based Search for Public Transit research paper [8]. The approach discussed in this paper suggests a process which finds arithmetic progressions with equal gap values. The algorithm starts with the minimum number and progresses forward while searching for the other numbers which have similar gap values. In this approach, the arithmetic progressions should always start with a number which is not contained in any other arithmetic progressions. However, within the progressions themselves, it is fine to have numbers which are already covered by other arithmetic progressions. According to the paper mentioned above, this approach is called covered by arithmetic progressions.

An improved version of this algorithm is also explained in the paper, where the aforementioned algorithm is run for several rounds, to find the arithmetic progressions with minimum length. The improved version of the algorithm yields good results by finding the longer arithmetic progressions over short arithmetic progressions. The research paper suggests the process that can be used to extract arithmetic progressions, but none of the implementation details is described there. Both of these algorithms and the approach is taken towards their implementation are discussed in the implementation chapter.

Metro map drawing is discussed by various research papers including the new algorithm for automatic visualization of metro map [19] paper and the metro map layout problem [20] paper. However, this domain is well addressed by the Metro Map Drawing Diploma thesis of Martin Nöllenburg [21]. The main idea proposed by this thesis is automatically generating the metro maps using an octilinear layout so that unnecessary geographical information and other irrelevant information will be ignored. This will help the commuters to focus on real questions like, which station should I get out, when is my transition, what is my current location etc. Octilinear layout is a layout with straight lines with horizontal, vertical and diagonals at 45 degrees. This layout makes the metro map easy to read and allows for quicker

extraction of information. The approach used here is to introduce hard constraints and some soft constraints where the hard constraints should be satisfied, and the soft constraints should be optimized. In this approach, metro map drawing is interpreted as a mixed integer problem, and then it addresses how to solve the problem. The code used to generate the metro map is not publicly available. The running time suggested by this approach is which is a global optimization problem is huge. Specifically, it can run from hours to even days to solve the optimization problem and the complexity of the optimization problem is NP-hard. Because of the proprietariness of the code and not being available to the public, it was not possible to try this approach or the tool developed under the research mentioned above. Therefore we tried other existing tools and software which are capable of drawing frequency graphs. By looking at the results output by the metro map drawing tool, it can be identified few scenarios that sign about some problem which could appear if we use this tool to render frequency graphs. One problem with the metro maps rendered by the metro map drawing tool is that it handles the multi-edges between two nodes poorly. Specifically, it renders multiple edges between stations as multi-colour single edges. This is not acceptable since we need individual edges for each frequency line between nodes.



**Figure 3:** Metro map generated by metro map drawing tool on Karlsruhe tram system [21]



As it is seen on the map, multi-edges rendered as single edges, and this makes it impossible to draw travel frequencies on the edges. This map is perfectly fine for its intended purpose which is as a metro map, but not the best fit for the drawing of the frequency graphs.

## 4 Implementation

Our main aim is to automatically generate a frequency map using public transit data. The public transit data is in GTFS format, and we have explicitly selected Swiss railway GTFS data for the implementation. The reasons for selecting the Swiss railway GTFS dataset are the consistency and the completeness of the dataset. Switzerland is a small country, and their railway infrastructure covers almost every part of the country with a high number of connections. Most of these connections are consistent during the working days of the week. That means the departure times and arrival times do not have differences at least during the working days. The exceptions are comparably low, and this is an advantage when finding the frequencies between nodes since this would allow us to generate frequency maps which are valid even for an entire month. Another factor for selecting the Swiss GTFS data is its availability. This data set is publicly available. Anyone can download the Swiss GTFS data set via their portal [22].

Switzerland railway agency has already generated a state of the art frequency map on their railway network. This is drawn manually, using graphical tools. Generating such graph manually from scratch is time-consuming and requires a lot of effort and resources. Availability of such a map is also a plus point here since we can use that frequency map to cross-check and validate our automatically generated frequency map and check the quality and drawbacks of the automatically generated frequency map. Another major advantage of our tool is the ability to generate frequency maps significantly faster than manually drawing them. It only needs few minutes to render a frequency map on a given date. This allows us to render frequency maps on a monthly basis, weekly basis and even on a daily basis. The advantage of that is, whenever we have a large number of additional connections (ex: during Christmas) and different schedules, it is possible to generate a frequency map within minutes.

Even though in our first implementation we focus on a country level, it is entirely possible to implement frequency maps at the state level, and city level as long as accurate GTFS data sets are available. Basically, frequency maps can be generated for tram network as well as bus network or any other public transport network. At the city level, it is more convenient to presents the frequency map alongside with the metro map, so that commuters would be able to get a clear understanding about how frequently each part of the cities is covered with

public transport. The speciality of the frequency maps is they are easy to read and easy to extract information from, as opposed to general timetables. Additionally, frequency maps can also give some information on the geographic location of the stations and the connection type between nodes (Intercity Express, Regional Express.)

Generating of frequency map for public transit data can be categorized into four main steps.

1. Extraction of transit graph from GTFS data.
2. Implementation of Frequency Finding Algorithm.
3. Extraction of the frequencies among interesting nodes and create frequency lines.
4. Drawing of frequency lines and nodes in a convenient way to resembles a schematic map.

Out of these four step process, the drawing of frequency lines and nodes to resemble a schematic map is the most challenging part. Because the map should be convenient, readable and node locations should satisfy the geographical location of the real stations. Even though the purpose of this thesis is not to generate a ready to print map, but at least it should help map makers by giving the relief of doing everything from scratch.

## **4.1 Extraction of transit graph from GTFS data**

A General Transit Feed Specification is a collection of series of text files which are collected in a ZIP format. This is a combination of different comma separated files. Some of these CSV files are mandatory for a GTFS feed, while others are optional. All these files and their purposes are listed in Google GTFS reference page as follows.

Filename	Required	Defines
<a href="#">agency.txt</a>	Required	One or more transit agencies that provide the data in this feed.
<a href="#">stops.txt</a>	Required	Individual locations where vehicles pick up or drop off passengers.
<a href="#">routes.txt</a>	Required	Transit routes. A route is a group of trips that are displayed to riders as a single service.
<a href="#">trips.txt</a>	Required	Trips for each route. A trip is a sequence of two or more stops that occurs at specific time.
<a href="#">stop_times.txt</a>	Required	Times that a vehicle arrives at and departs from individual stops for each trip.
<a href="#">calendar.txt</a>	Required	Dates for service IDs using a weekly schedule. Specify when service starts and ends, as well as days of the week where service is available.
<a href="#">calendar_dates.txt</a>	Optional	Exceptions for the service IDs defined in the <a href="#">calendar.txt</a> file. If <a href="#">calendar.txt</a> includes ALL dates of service, this file may be specified instead of <a href="#">calendar.txt</a> .
<a href="#">fare_attributes.txt</a>	Optional	Fare information for a transit organization's routes.
<a href="#">fare_rules.txt</a>	Optional	Rules for applying fare information for a transit organization's routes.
<a href="#">shapes.txt</a>	Optional	Rules for drawing lines on a map to represent a transit organization's routes.
<a href="#">frequencies.txt</a>	Optional	Headway (time between trips) for routes with variable frequency of service.
<a href="#">transfers.txt</a>	Optional	Rules for making connections at transfer points between routes.
<a href="#">feed_info.txt</a>	Optional	Additional information about the feed itself, including publisher, version, and expiration information.

**Figure 4:** Google GTFS reference page [2]

The basic process in building a transit graph is to read through all the mandatory GTFS files and store the necessary information of these files in data structures. Data structures should be designed in a way so that all later processes of generating a frequency map can be performed on this data structure without any issues. Java is the programming language that is used to implement the algorithm.

In order to understand how the algorithm works and to generate the transit graph using the GTFS data, it is necessary to have a deep look into the required CSV files of the GTFS data and data structures which are used to store extracted information.

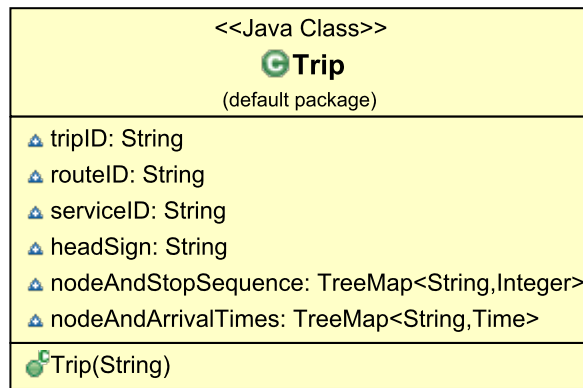
```

route_id,service_id,trip_id,trip_headsign,trip_short_name,direction_id,block_id,shape_id,bikes_allowed,attributes_ch
03002.06____:3002,1:1:s,1,Neckarbischofsheim Nord,3002,,,,,0,TS
03003.06____:3003,2:1:s,2,Untergimper,3003,,,,,0,TS
03005.06____:3005,3:1:s,3,Hüffenhardt,3005,,,,,0,TS
03006.06____:3006,4:1:s,4,Neckarbischofsheim Nord,3006,,,,,0,TS
03007.06____:3007,5:1:s,5,Hüffenhardt,3007,,,,,0,TS
03008.06____:3008,6:1:s,6,Neckarbischofsheim Nord,3008,,,,,0,TS
03009.06____:3009,7:1:s,7,Hüffenhardt,3009,,,,,0,TS
03012.06____:3012,091831,8,Neckarbischofsheim Nord,3012,,,,,0,
03013.06____:3013,091831,9,Hüffenhardt,3013,,,,,0,
03014.06____:3014,091831,10,Neckarbischofsheim Nord,3014,,,,,0,
03015.06____:3015,091831,11,Hüffenhardt,3015,,,,,0,
03016.06____:3016,091831,12,Neckarbischofsheim Nord,3016,,,,,0,

```

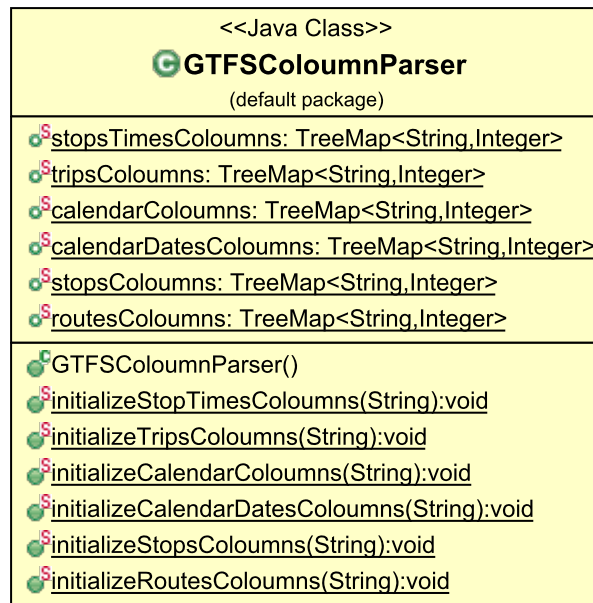
**Figure 5:** trips.txt [3]

The algorithm starts with reading the `trips.txt` file. It stores all the necessary information in the `trip.txt` file inside of a `TreeMap` variable. The key of the `TreeMap` is the trip id which is `String` and the value is a `Trip` instance. The `Trip` class is a data model which is used to store all the trip related data.



**Figure 6:** Class diagram of Trip class

The `trip.txt` file is read line by line by the algorithm, and the first line will read and stored as a map called `tripsColumns` inside `GTFSColoumnParser`. The key of this map is the header attribute, and the value is the index of the header attribute. This map becomes useful when reading further lines and extracting their information and creating the trip objects. This approach is used, when reading all other GTFS files, since by using this mechanism it is possible to minimize the problems which occur when the format of the header attributes slightly changed from each GTFS feeds.



**Figure 7:** Class diagram of GTFSColoumnParser class

As per the Trip class diagram, two additional maps are used to store the more details about the trips.

- nodeAndStopSequence

This map is used to keep the record of the nodes covered by a trip and its sequence.

- nodeAndArrivalTimes

This map keeps the record of the nodes and the arrival time to those nodes during the trip.

Information stored in these maps become really useful later, specifically when finding the frequency between two distantly located nodes. Because it helps to determine the order of the nodes within the trip and at which time each node is reached during the journey.

```

route_id,agency_id,route_short_name,route_long_name,route_desc,route_type,route_url,route_color,route_text_color
03002.06____:3002,06____,RB,RB 3002,,2,,,
03003.06____:3003,06____,RB,RB 3003,,2,,,
03005.06____:3005,06____,RB,RB 3005,,2,,,
03006.06____:3006,06____,RB,RB 3006,,2,,,
03007.06____:3007,06____,RB,RB 3007,,2,,,
03008.06____:3008,06____,RB,RB 3008,,2,,,
03009.06____:3009,06____,RB,RB 3009,,2,,,
03012.06____:3012,06____,RB,RB 3012,,2,,,
03013.06____:3013,06____,RB,RB 3013,,2,,,
03014.06____:3014,06____,RB,RB 3014,,2,,,
03015.06____:3015,06____,RB,RB 3015,,2,,,
03016.06____:3016,06____,RB,RB 3016,,2,,,

```

Figure 8: routes.txt [3]

After finishing the extracting of the `trip.txt` file, the program then reads the `routes.txt` file. Route information is also stored in a `TreeMap` variable. Route id is used as a key and the value is a `Route` instance. `Route` class is used as the data model to store all the route related data.

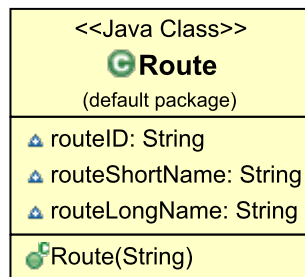


Figure 9: Class diagram of `Route` class

`Routes` consists of route type and when finding the color for the frequency lines, this information is needed.

```

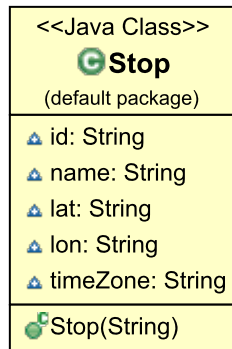
|stop_id,stop_code,stop_name,stop_desc,stop_lat,stop_lon,stop_elevation,zone_id,stop_url,location_type,parent_station,platform_code,ch_station_long_name,ch_station_synonym1,ch_station_synonym2,ch_station_synonym3,ch_station_synonym4
8504012:1,PUI,Puidoux-Chexbres,,46.493833,6.765686,618,,,0,8504012,1,,,,
8504012:2,PUI,Puidoux-Chexbres,,46.493833,6.765686,618,,,0,8504012,2,,,,
132,,Bahn-2000-Strecke,,47.196374,7.689360,0,,,0,,,Bahn-2000-Strecke,,,
133,,Centovalli,,46.154371,8.603653,0,,,0,,,Centovalli,,,
134,,Furka,,46.538322,8.435913,0,,,0,,,Furka,,,
135,,Lötschberg-Basistunnel,,46.356888,7.773846,0,,,0,,,Lötschberg-Basistunnel,,,
136,,Lötschberg-Bergstrecke,,46.433756,7.717215,0,,,0,,,Lötschberg-Bergstrecke,,,
137,,Rotsee,,47.074530,8.320797,0,,,0,,,Rotsee,,,
138,,Vereina,,46.808381,9.985688,0,,,0,,,Vereina,,,
139,,Damm,,47.217327,8.805061,0,,,0,,,Damm,,,
140,,Gotthard-Panoramastrecke,,46.604329,8.591156,0,,,0,,,Gotthard-Panoramastrecke,,,
141,,Oberalp,,46.652359,8.644440,0,,,0,,,Oberalp,,,
142,,Simplon,,46.276342,8.091198,0,,,0,,,Simplon,,,
8502204:6CD,ZG,Zug,,47.173624,8.515295,425,,,0,8502204,6CD,,,,

```

Figure 10: stops.txt [3]

The `stops.txt` file consists with all the node related details. Node information is stored in a `TreeMap` variable. Node id is used as a key and `Stop` instances are used as the value. `Stop`

class is used as a data model to store the information of nodes.



**Figure 11:** Class diagram of Stop class

Node name, latitude and longitude are used to draw the nodes when generating the frequency map. Nodes are identified by node id inside the frequency finding algorithm.

```

service_id,monday,tuesday,wednesday,thursday,friday,saturday,sunday,start_date,end_date
000000,1,1,1,1,1,1,1,20161211,20171209
|

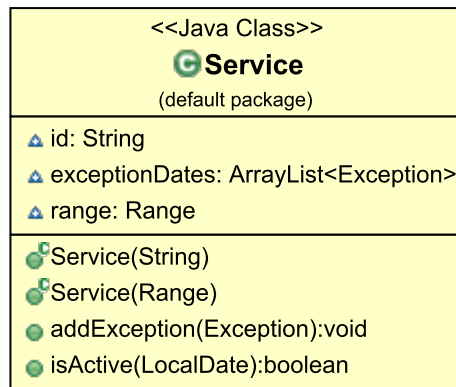
```

**Figure 12:** calender.txt [3]

Normally `calendar.txt` should contain the starting and end time of the services for a given year as well as available days of services. In a normal GTFS feed, all of the services that are available should be included in `calendar.txt`. The Swiss GTFS dataset does not contain any information, and it uses `calendar_dates.txt` file to mention exception dates of the services. That means that in the Swiss GTFS dataset, `calendar_dates.txt` file consists of all the service ids which are mapped with trips.

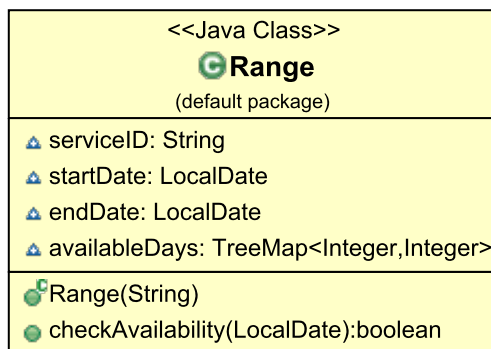
Nevertheless, in order to make the program work with any given GTFS data, the program goes through the `calendar.txt` file and stores all of the service information within a Treemap. Service id is used as the key and instances of Service class are used as values. The Service class is used as the data model to store the service related data including service id, exception dates, and starting and end date of the service.





**Figure 13:** Class diagram of Service class

In order to store the available dates and the start and end date of service, another data model called Range is used and Service instances consist a Range attribute.



**Figure 14:** Class diagram of Range class

```

service_id,date,exception_type
10704:4:4:s,20171202,1
25358:1:s,20170626,1
25358:1:s,20170627,1
25358:1:s,20170625,1
25358:1:s,20171211,1
25358:1:s,20171210,1
25358:1:s,20170628,1
25358:1:s,20170629,1
25358:1:s,20170703,1
25358:1:s,20170702,1
  
```

**Figure 15:** calendar\_dates.txt [3]

calendar\_dates.txt file is not a mandatory file for GTFS data feed. However, in most of the GTFS feeds, it can be found because exceptions are normal scenarios in a public transit network. When a service is deviating from its standard dates as defined in the calendar.txt,

it is included in the `calendar_dates.txt` file with relevant exception type. It is possible to use `calendar_dates.txt` file to both activate and to deactivate service on a given date.

The program will go through the `calendar_dates.txt` file and add all exception dates to the services and updates all the services. Recording of exception dates allows the algorithm to find the frequency lines between nodes, for a given date. This can be useful when it is needed to generate frequency map for weekdays as well as weekends.

```
trip_id,arrival_time,departure_time,stop_id,stop_sequence,stop_headsign,pickup_type,drop_off_type,shape_dist_traveled,attributes_ch
1,11:42:00,11:42:00,8050807,0,,0,0,,
1,11:46:00,11:46:00,8050806,1,,0,0,,
1,11:52:00,11:52:00,8050805,2,,0,0,,
1,11:56:00,11:56:00,8050804,3,,0,0,,
1,11:59:00,11:59:00,8050803,4,,0,0,,
1,12:05:00,12:05:00,8050802,5,,0,0,,
1,12:12:00,12:12:00,8091916,6,,0,0,,
2,12:20:00,12:20:00,8091916,0,,0,0,,
2,12:24:00,12:24:00,8050802,1,,0,0,,
2,12:28:00,12:28:00,8050803,2,,0,0,,
2,12:32:00,12:32:00,8050804,3,,0,0,,
3,13:32:00,13:32:00,8050804,0,,0,0,,
3,13:37:00,13:37:00,8050805,1,,0,0,,
```

Figure 16: `stop_times.txt` [3]

`stop_times.txt` consists of all the schedules of the given transit network. Arrival times as well as departure times from nodes are included here and the stop sequence of a particular node within a trip can also be found here. The file is read line by line and instances of Node are created and stored in nodeCollection map. nodeCollection is a TreeMap variable which stores node ids as keys and instances of Node class as values. Node is a data model class that is used to store the outConnections and inConnections of the node and the trip ids of the trips which cover this node in their journey.

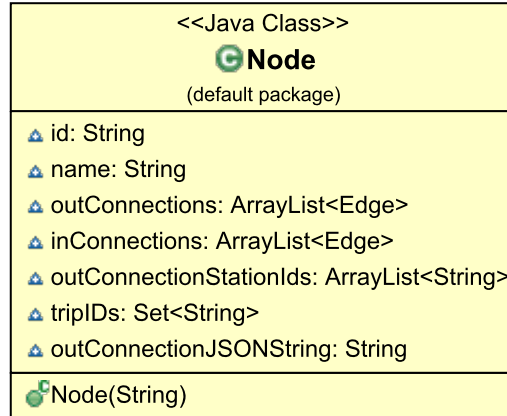
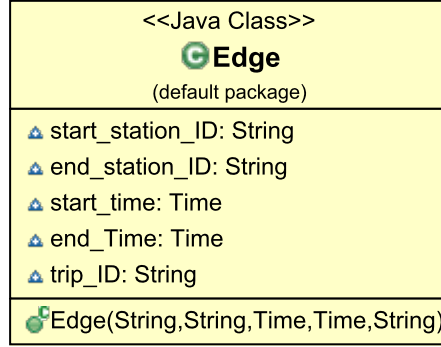


Figure 17: Class diagram of Node class

By now all the nodes of the transit feed are stored in nodeCollection TreeMap. As mentioned above, every node consists of inConnections as well as outConnections attributes. These

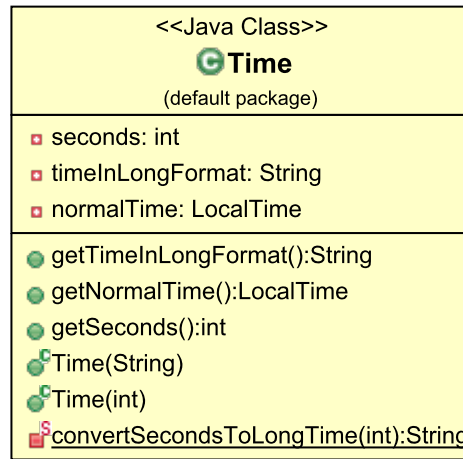
attributes are the collection of Edge type. Edges are used to represents the arriving and leaving journeys to a given node.

- inConnections:Arriving journeys to a node.
- outConnection:Departing journeys from a node.



**Figure 18:** Class diagram of Edge class

An edge also records the starting time from the starting node and the arrival time to the destination node. Arrival time and departure times are stored as Time instances. Time instances are specially designed to store time in seconds. The reason behind being that is it is more convenient to store and use time as a single integer when finding the frequencies between a set of departure times.



**Figure 19:** Class diagram of Time class

When reading of `stop_times.txt` finishes, the entire transit network is created and stored in the memory. All of the arrivals and all of the departures for every node is now successfully

stored in data structures. Using this structure it is possible to check which nodes are consecutively located to a given node by analysing the inConnections and outConnections of that node. Then, the program writes the transit graph into Graph.txt, so that it is possible to see and analyse the transit graph that is created using the GTFS data. This transit graph will be the base source of the pipeline process of automatically generating the frequency maps.

#### Graph Structure

Node: 8000339

##### In connections

```
Start Station 8002238 arrival Time 00:04:00 Trip ID: 8582:1, Service ID: 8582:1:1:s
Start Station 8002238 arrival Time 00:04:00 Trip ID: 8582:2, Service ID: 8582:2:2:s
Start Station 8002238 arrival Time 00:04:00 Trip ID: 8582:3, Service ID: 8582:3:3:s
Start Station 8000988 arrival Time 04:52:00 Trip ID: 8583, Service ID: 8583:1:s
Start Station 8000988 arrival Time 04:53:00 Trip ID: 8584, Service ID: 8584:1:s
Start Station 8002238 arrival Time 05:04:00 Trip ID: 8585, Service ID: 8585:1:s
```

##### Out connections

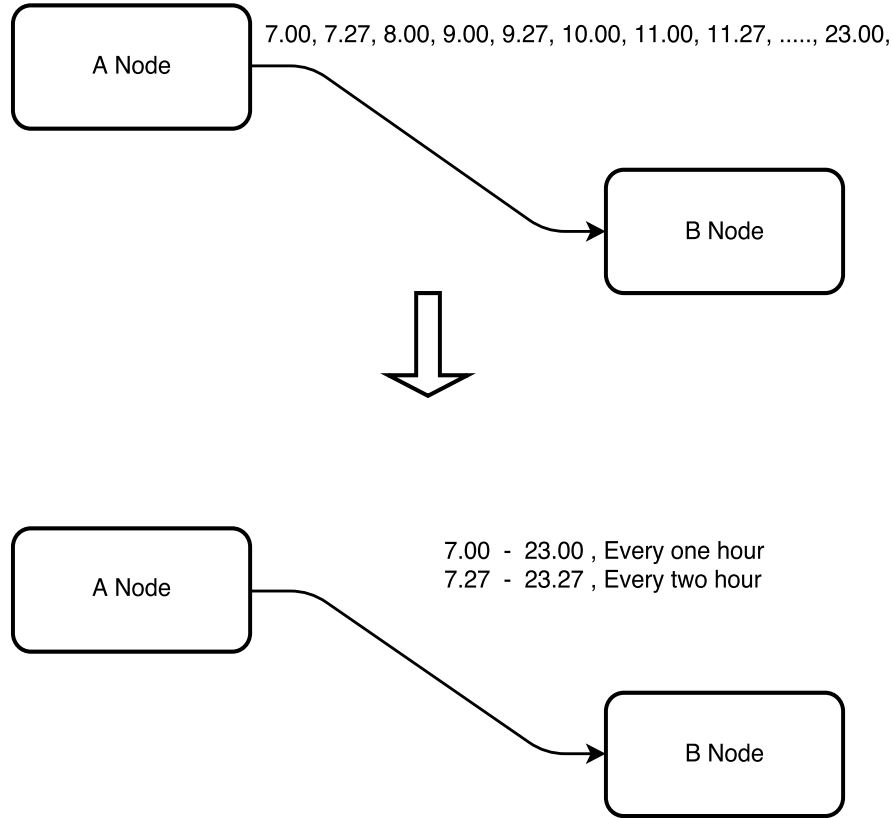
```
End Station 8000988 depart Time 00:04:00 Trip ID: 8582:1, Service ID: 8582:1:1:s
End Station 8000988 depart Time 00:04:00 Trip ID: 8582:2, Service ID: 8582:2:2:s
End Station 8000988 depart Time 00:04:00 Trip ID: 8582:3, Service ID: 8582:3:3:s
End Station 8002238 depart Time 04:52:00 Trip ID: 8583, Service ID: 8583:1:s
End Station 8002238 depart Time 04:53:00 Trip ID: 8584, Service ID: 8584:1:s
End Station 8000988 depart Time 05:04:00 Trip ID: 8585, Service ID: 8585:1:s
End Station 8002238 depart Time 05:53:00 Trip ID: 8586:1, Service ID: 8586:1:1:s
```

**Figure 20:** Transit graph built by reading GTFS dataset

## 4.2 Frequency finding algorithm.

### 4.2.1 Implementation of Frequency Finding Algorithm

The main objective of the process of frequency finding is the finding of arithmetic progressions inside a number sequence. Most of the public transport mediums operate in a periodical manner. For the simplicity let's consider two consecutive nodes A and B. Assume there are sets of departures from node A to node B from the beginning of the day until the end of the day.



**Figure 21:** Departure times between Node A and B

In above example the frequencies of the departures can be listed as follows

1. 7:00 - 23:00, once every hour
2. 7:27 - 23:27, once every two hour

By identifying these frequencies, it is also possible to compress all the departure times from node A to node B, into two frequency label. Rather than storing all the departure times, it is more convenient to store them as frequency labels. Because of that, the space required to store these departure times is minimized.

This approach is well explained in the Frequency-Based Search for Public Transit [8] research paper. The core of the approach suggested by the paper is used to find the departure time frequency between two nodes. However, finding the frequency of a number sequence is not always that trivial, the reason for this is, in real life it is possible to have hundreds of departure times from a starting node. In such a case, there might be a large number of hidden frequencies inside these departure times. Finding all of these frequencies manually is nearly impossible

since the higher the number of departure times, the higher the possibility of existing hidden frequencies among these departures. The algorithm suggested by the paper consists of the following steps.

1. Start with the smallest number  $t$  in the number sequence and then search for the longest arithmetic progression which starts with this number.
2. Once the longest arithmetic progression is found, the algorithm then adds this arithmetic progression into a collection and marks all the numbers in that arithmetic progression as covered.
3. The algorithm then starts with the next smallest number which is not marked as covered and progresses as step 1 and 2.

The algorithm does not start the arithmetic progression with numbers which are already marked as covered. Rather, it allows the covered numbers to be included in other arithmetic progressions. This means one element can be included in two or more arithmetic progressions. This is known as cover by arithmetic progressions (CAP). When selecting the best arithmetic progression among two progressions, the algorithm always select the arithmetic progression with the most uncovered elements.

To understand the algorithm suggested by the paper let us consider a sample number sequence.

$$T = \{ 2, 6, 12, 18, 24, 36, 48, 60, 70, 72, 84 \}$$

The algorithm starts a search for arithmetic progressions with number 2, and the longest arithmetic progression that it can find is

$$\{ 2, 36, 70 \}$$

Then it starts with 6 and the next arithmetic progression that it finds is

$$\{ 6, 12, 18, 24 \}$$

Next, it starts with 48 since 12 is already included in an arithmetic progression. The elements which are already covered by other arithmetic progressions cannot be the prefixes of another progression as mentioned previously. Therefore the next arithmetic progression is

$$\{ 48, 60, 72, 84 \}$$

The runtime of this algorithm is  $O(T^3)$ . The arithmetic progressions found by this algorithm is suboptimal. That means, it is possible to find longer arithmetic progression, when the algorithm is tweaked a bit. This is called as the improved frequency finding algorithm.

## Improved Frequency Finding Algorithm

This mechanism is also described in the paper as an improved version of the above approach. Instead of starting with the first element and searching for the longest arithmetic progression, the new approach executes the frequency finding algorithm for multiple rounds to find the arithmetic progressions with minimum length  $K$ . Initially this  $K$  is a high number and then reduce it in an iterative manner. In each iteration, it tries to find an arithmetic progression which satisfies the minimum length condition. Let us consider the number sequence again.

$$T = \{ 2, 6, 12, 18, 24, 36, 48, 60, 70, 72, 84 \}$$

Starting with  $K = 7$ , the algorithm finds the arithmetic progression of

$$\{ 12, 24, 36, 48, 60, 72, 84 \}$$

$K = 6$ , No arithmetic progression with length of 6.

$K = 5$ , No arithmetic progression with length of 5.

$K = 4$ , No arithmetic progression with length of 4.

$K = 3$ , two arithmetic progressions with length of 3.

$$\{ 6, 12, 18 \}$$

$$\{ 2, 36, 70 \}$$

The approach described in the paper does not cover the situation in which of resulting in two arithmetic progressions which have the same length. This needs to be sorted out since it is really important when there are two arithmetic progressions with the same length which consists of the same number of elements and share one or more elements with each other. In this situation which arithmetic progress should be selected?.

The approach used in our algorithm is such that, in a situation like this, the algorithm compares the number of uncovered elements that are covered by each arithmetic progression. The arithmetic progression with the most uncovered elements will be chosen first, and if the two arithmetic progressions are with the same number of uncovered elements, then it will choose the arithmetic progression with the minimum gap sum. In this way, the program prioritizes the selection of arithmetic progression when there are multiple arithmetic progressions to be selected.

In this improved version of the frequency finding algorithm, the value for  $K$  can be used as the length of the input number sequence. Yet, in most situations this is not efficient. The

reason for that is the probability of finding an arithmetic progression which has the length as long as the number sequence is significantly low. This is because in most cases the number sequence consists of multiple arithmetic progressions. As soon as it consists of more than one arithmetic progression, then the maximum length of the arithmetic progressions has to be less than the length of the input number sequence. Using a high value for K will unnecessarily increase the running time of the algorithm drastically, since each time the algorithm will run through the number sequence trying to find an arithmetic progression of the minimum of K length. So the value of K is heuristically based on the GTFS feed.

When implementing the improved version of the frequency finding algorithm, the GTFS feed of the streetcars in Basel is used. Streetcars have significantly high frequencies when comparing them to trains, and this allowed us to test whether the frequency finding algorithm accurately detects all the available frequencies. For this GTFS feed, the following heuristic is used to decide the initial value of K in order to reduce the running time of the frequency finding algorithm while not degrading the arithmetic progressions it discovers.

---

**Algorithm 1** Algorithm for decide initial value for K, pseudo code :

---

```

1 k = 0
2 if length of number sequence > 800 then
3   k = length of number sequence / 4
4 else if length of number sequence > 600 then
5   k = length of number sequence / 3
6 else if length of number sequence > 450 then
7   k = length of number sequence / 2
8 else if length of number sequence > 300 then
9   k = length of number sequence - (length of number sequence / 3)
10 else
11   k = length of number sequence

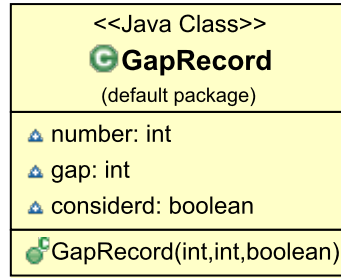
```

---

The same heuristic is adapted for the Swiss railway GTFS feed and most of the cases the departure time sequences which are submitted for the frequency finding algorithm have a length of less than 300. So the frequency finding algorithm runs for a number of iterations of the full length of the departure time sequence.

Implementation of the frequency finding algorithm is done using Java language. Here, a custom class called GapRecord is used to store the numbers of the input sequences. Each number in the input sequence is stored in an instance of GapRecord, and the gap value to the next number in the sequence is also stored inside the GapRecord.





**Figure 22:** Class diagram of GapRecord class

When all the GapRecords objects are created, then, the frequency finding algorithm starts executing.

---

**Algorithm 2** Algorithm for frequency finding is as follows.

---

```

1  ArrayList<ArrayList<GapRecord>> seriesStartsWithRecordX
2  GapRecord currentRecord
3
4  for (int x = 0; x < records.size(); x++) {
5      if (records.get(x).considerd == false) {
6          seriesStartsWithRecordX = new ArrayList<ArrayList<GapRecord>>();
7          seriesStartsWithRecordX.add(new ArrayList<GapRecord>());
8          currentRecord = records.get(x);
9          seriesStartsWithRecordX.get(seriesStartsWithRecordX.size() - 1).add(
              currentRecord);
10         generateSequence(records.get(x).gap, x + 1, 0);
11     }
12 }
```

---

The code mentioned in algorithm 2 will go through the collection called records. Records is a list of GapRecord instances which is created based on the input number sequence. During each iteration it initializes a collection named seriesStartsWithRecordX in order to store the arithmetic progressions started with the xth item of the records collection. currentRecord variable is used to hold a reference to the xth item of the records. Then it calls the generateSequence() method to find the arithmetic progressions starting with the xth item of the records collection.

---

**Algorithm 3** Algorithm for finding the arithmetic progressions starts with the xth item of the records collection.

---

```
1 void generateSequence(long expectedGap, int nextIndex, long currentGap)
2
3 long gap = currentGap + records.get(nextIndex - 1).gap
4 if (expectedGap == gap) {
5     seriesStartsWithRecordX.get(seriesStartsWithRecordX.size() - 1).add(
6         records.get(nextIndex))
7
8     if ((nextIndex + 1) < records.size()) {
9         generateSequence(expectedGap, (nextIndex + 1), 0)
10    }
11 } else if (expectedGap > gap) {
12     if ((nextIndex + 1) < records.size()) {
13         generateSequence(expectedGap, nextIndex + 1,
14             (records.get(nextIndex - 1).gap + currentGap));
15     }
16 } else if (expectedGap < gap) {
17     if ((nextIndex + 1) < records.size()) {
18         int currentRecordIndex = records.indexOf(currentRecord)
19         int nxtrec = currentRecordIndex + seriesStartsWithRecordX.size() +
20             1
21         seriesStartsWithRecordX.add(new ArrayList<GapRecord>())
22         seriesStartsWithRecordX.get(seriesStartsWithRecordX.size()-1).add(
23             currentRecord)
24
25         int totalGapSumm = 0
26         for (int i = currentRecordIndex; i < (nxtrec); i++) {
27             totalGapSumm += records.get(i).gap
28         }
29         generateSequence(totalGapSumm, nxtrec + 1, 0)
30    }
31 }
```

---

generateSequence() method takes expectedGap, nextIndex and currentGap as parameters.

- expectedGap: The gap value which the algorithm is expecting to have between numbers.
- nextIndex: Index for the next element in records collection
- currentGap: The sum of the gap values of the gap records

generateSequence() method is a recursive method; once it is called with a gapRecord and

other necessary parameters, the function will recursively call itself until the end of the number sequence for finding arithmetic progressions starting with the number which is inside the initially passed gapRecord. The process behind the frequency finding is a gap finding mechanism. The algorithm progresses forward to find the equal gaps between numbers. If the expectedGap is higher than the current gap, it calls the generateSequence() function recursively with the next index and the currentGap value. If the expectedGap is lower than the currentGap value, then it again calls for the generateSequence() method, with the new value for the expectedGap and the nextIndex. The identified arithmetic progressions are stored inside the seriesesStartsWithRecordX collection.

Once this step is completed the identified arithmetic progressions should be filtered. The algorithm first removes the already considered elements from the arithmetic progressions if they are included as suffixes. Then the number of uncovered/ unconsidered elements is counted, and the sum of the gap values of each sequence is calculated. Then the algorithm selects the arithmetic progressions which meets the minimum length requirement(K) and list them as eligible arithmetic progressions. When the number of uncovered elements are the same in two arithmetic progressions, and if they share one or more elements between their arithmetic progressions, then the algorithm selects the progression with the minimum sum of the gap values. This way, the algorithm selects the arithmetic progressions given a number sequence.

Since the frequency finding algorithm works on numbers, all the departure times are converted into seconds before they feed into the frequency finding algorithm. We adapted this algorithm to find the arithmetic progressions among departure times between nodes, in a public transit network.

We first evaluate the frequency finding algorithm via the Swiss street cars GTFS dataset, to check the results the algorithm outputs.

## Frequency Covers on: 2017-01-01

Time Series	Frequency Labels		
Time Series:(10:57:00, 11:02:00, 11:07:00, 11:12:00, 11:17:00, 11:22:00, 11:27:00, 11:32:00, 11:37:00, 11:42:00, 11:47:00, 11:52:00, 11:57:00, 12:02:00, 12:07:00, 12:12:00, 12:17:00, 12:22:00, 12:27:00, 12:32:00, 12:37:00, 12:42:00, 12:47:00, 12:52:00, 12:57:00, 13:02:00, 13:07:00, 13:12:00, 13:17:00, 13:22:00, 13:27:00, 13:32:00, 13:37:00, 13:42:00, 13:47:00, 13:52:00, 13:57:00, 14:02:00, 14:07:00, 14:12:00, 14:17:00, 14:22:00, 14:27:00, 14:32:00, 14:37:00, 14:42:00, 14:47:00, 14:52:00, 14:57:00, 15:02:00, 15:07:00, 15:12:00, 15:17:00, 15:22:00, 15:27:00, 15:32:00, 15:37:00, 15:42:00, 15:47:00, 15:52:00, 15:57:00, 16:02:00, 16:07:00, 16:12:00, 16:17:00, 16:22:00, 16:27:00, 16:32:00, 16:37:00, 16:42:00, 16:47:00, 16:52:00, 16:57:00, 17:02:00, 17:07:00, 17:12:00, 17:17:00, 17:22:00, 17:27:00, 17:32:00, 17:37:00, 17:42:00, 17:47:00, 17:52:00, 17:57:00, 18:02:00, 18:07:00, 18:12:00, 18:17:00, 18:22:00, 18:27:00, 18:32:00, 18:37:00, 18:42:00, 18:47:00, 18:52:00, 18:57:00, 19:02:00, 19:07:00, 19:12:00, 19:17:00, 19:22:00, 19:27:00, 19:32:00)	Start Time	End Time	Frequency
	10:57:00	19:32:00	5 minutes
Time Series:(05:32:00, 05:47:00, 06:02:00, 06:17:00, 06:32:00, 06:47:00, 07:02:00, 07:17:00, 07:32:00, 07:47:00, 08:02:00, 08:17:00, 08:32:00, 08:47:00, 09:02:00, 09:17:00, 09:32:00, 09:47:00, 10:02:00, 10:17:00, 10:32:00, 10:47:00, 11:02:00, 11:17:00, 11:32:00, 11:47:00, 12:02:00, 12:17:00, 12:32:00, 12:47:00, 13:02:00, 13:17:00, 13:32:00, 13:47:00, 14:02:00, 14:17:00, 14:32:00, 14:47:00, 15:02:00, 15:17:00, 15:32:00, 15:47:00, 16:02:00, 16:17:00, 16:32:00, 16:47:00, 17:02:00, 17:17:00, 17:32:00, 17:47:00, 18:02:00, 18:17:00, 18:32:00, 18:47:00, 19:02:00, 19:17:00, 19:32:00, 19:47:00, 20:02:00, 20:17:00, 20:32:00, 20:47:00, 21:02:00, 21:17:00, 21:32:00, 21:47:00, 22:02:00, 22:17:00, 22:32:00, 22:47:00, 23:02:00, 23:17:00, 23:32:00, 23:47:00, 24:02:00, 24:17:00, 24:32:00)	Start Time	End Time	Frequency
	05:32:00	24:32:00	15 minutes
Time Series:(11:04:00, 11:14:00, 11:24:00, 11:34:00, 11:44:00, 11:54:00, 12:04:00, 12:14:00, 12:24:00, 12:34:00, 12:44:00, 12:54:00, 13:04:00, 13:14:00, 13:24:00, 13:34:00, 13:44:00, 13:54:00, 14:04:00, 14:14:00, 14:24:00, 14:34:00, 14:44:00, 14:54:00, 15:04:00, 15:14:00, 15:24:00, 15:34:00, 15:44:00, 15:54:00, 16:04:00, 16:14:00, 16:24:00, 16:34:00, 16:44:00, 16:54:00, 17:04:00, 17:14:00, 17:24:00, 17:34:00, 17:44:00, 17:54:00, 18:04:00, 18:14:00, 18:24:00, 18:34:00, 18:44:00, 18:54:00, 19:04:00)	Start Time	End Time	Frequency
	11:04:00	19:04:00	10 minutes
Time Series:(11:10:00, 11:20:00, 11:30:00, 11:40:00, 11:50:00, 12:00:00, 12:10:00, 12:20:00, 12:30:00, 12:40:00, 12:50:00, 13:00:00, 13:10:00, 13:20:00, 13:30:00, 13:40:00, 13:50:00, 14:00:00, 14:10:00, 14:20:00, 14:30:00, 14:40:00, 14:50:00, 15:00:00, 15:10:00, 15:20:00, 15:30:00, 15:40:00, 15:50:00, 16:00:00, 16:10:00, 16:20:00, 16:30:00, 16:40:00, 16:50:00, 17:00:00, 17:10:00, 17:20:00, 17:30:00, 17:40:00, 17:50:00, 18:00:00, 18:10:00, 18:20:00, 18:30:00, 18:40:00, 18:50:00, 19:00:00)	Start Time	End Time	Frequency
	11:10:00	19:00:00	10 minutes

Figure 23: Set of identified frequency covers between Basel Bankverein - Basel Aeschenplatz

Time Series:(05:14:00, 05:29:00, 05:44:00, 05:59:00, 06:14:00, 06:29:00, 06:44:00, 06:59:00, 07:14:00, 07:29:00, 07:44:00, 07:59:00, 08:14:00, 08:29:00, 08:44:00, 08:59:00, 09:14:00, 09:29:00, 09:44:00, 09:59:00, 10:14:00, 10:29:00, 10:44:00, 10:59:00)	<b>Start Time</b>	<b>End Time</b>	<b>Frequency</b>
	05:14:00	10:59:00	15 minutes
Time Series:(05:18:00, 05:33:00, 05:48:00, 06:03:00, 06:18:00, 06:33:00, 06:48:00, 07:03:00, 07:18:00, 07:33:00, 07:48:00, 08:03:00, 08:18:00, 08:33:00, 08:48:00, 09:03:00, 09:18:00, 09:33:00, 09:48:00, 10:03:00, 10:18:00, 10:33:00, 10:48:00)	<b>Start Time</b>	<b>End Time</b>	<b>Frequency</b>
	05:18:00	10:48:00	15 minutes
Time Series:(04:50:00, 05:05:00, 05:20:00, 05:35:00, 05:50:00, 06:05:00, 06:20:00, 06:35:00, 06:50:00, 07:05:00, 07:20:00, 07:35:00, 07:50:00, 08:05:00, 08:20:00, 08:35:00, 08:50:00, 09:05:00, 09:20:00, 09:35:00, 09:50:00, 10:05:00)	<b>Start Time</b>	<b>End Time</b>	<b>Frequency</b>
	04:50:00	10:05:00	15 minutes
Time Series:(05:40:00, 05:55:00, 06:10:00, 06:25:00, 06:40:00, 06:55:00, 07:10:00, 07:25:00, 07:40:00, 07:55:00, 08:10:00, 08:25:00, 08:40:00, 08:55:00, 09:10:00, 09:25:00, 09:40:00, 09:55:00, 10:10:00, 10:25:00, 10:40:00)	<b>Start Time</b>	<b>End Time</b>	<b>Frequency</b>
	05:40:00	10:40:00	15 minutes
Time Series:(06:08:00, 06:23:00, 06:38:00, 06:53:00, 07:08:00, 07:23:00, 07:38:00, 07:53:00, 08:08:00, 08:23:00, 08:38:00, 08:53:00, 09:08:00, 09:23:00, 09:38:00, 09:53:00, 10:08:00, 10:23:00, 10:38:00, 10:53:00, 11:08:00)	<b>Start Time</b>	<b>End Time</b>	<b>Frequency</b>
	06:08:00	11:08:00	15 minutes
Time Series:(19:29:00, 19:44:00, 19:59:00, 20:14:00, 20:29:00, 20:44:00, 20:59:00, 21:14:00, 21:29:00, 21:44:00, 21:59:00, 22:14:00, 22:29:00, 22:44:00, 22:59:00, 23:14:00, 23:29:00, 23:44:00, 23:59:00, 24:14:00, 24:29:00)	<b>Start Time</b>	<b>End Time</b>	<b>Frequency</b>
	19:29:00	24:29:00	15 minutes
Time Series:(19:33:00, 19:48:00, 20:03:00, 20:18:00, 20:33:00, 20:48:00, 21:03:00, 21:18:00, 21:33:00, 21:48:00, 22:03:00, 22:18:00, 22:33:00, 22:48:00, 23:03:00, 23:18:00, 23:33:00, 23:48:00, 24:03:00, 24:18:00)	<b>Start Time</b>	<b>End Time</b>	<b>Frequency</b>
	19:33:00	24:18:00	15 minutes
Time Series:(19:35:00, 19:50:00, 20:05:00, 20:20:00, 20:35:00, 20:50:00, 21:05:00, 21:20:00, 21:35:00, 21:50:00, 22:05:00, 22:20:00, 22:35:00, 22:50:00, 23:05:00, 23:20:00, 23:35:00, 23:50:00, 24:05:00, 24:20:00)	<b>Start Time</b>	<b>End Time</b>	<b>Frequency</b>
	19:35:00	24:20:00	15 minutes
Time Series:(19:53:00, 20:08:00, 20:23:00, 20:38:00, 20:53:00, 21:08:00, 21:23:00, 21:38:00, 21:53:00, 22:08:00, 22:23:00, 22:38:00, 22:53:00, 23:08:00, 23:23:00, 23:38:00, 23:53:00, 24:08:00, 24:23:00)	<b>Start Time</b>	<b>End Time</b>	<b>Frequency</b>
	19:53:00	24:23:00	15 minutes

**Figure 24:** Set of identified frequency covers between Basel Bankverein - Basel Aeschenplatz  
2

By observing the identified frequencies shown above, it is clear that streetcar schedules consist of significantly high-frequency covers. The real reason for having the streetcars in metropolitan areas is to facilitate the commuters who frequently travel among different sections of the city. Because of this, it is required to have a frequency coverage like this to meet the commuters

demand. This is only one set of the identified frequencies between these two consecutive nodes, and there are many other medium-sized identified frequencies that are not mentioned here.

ex: 08:10, 08:20, 08:30 . . . .

09:15, 09: 30, 09:45 . . . etc

Rather than checking for every possible gap values/ frequency inside the number sequence, it is more convenient to check for predefined frequencies like 10 minutes, 15 minutes, 20 minutes, 30 minutes etc. The advantage of this approach is that we can then extract the frequencies which are human-friendly and easy to remember for the commuters.

#### 4.2.2 Human-friendly frequency finding and filtering frequencies

The frequency finding algorithm reveals some frequencies which are harder to memorize as well as other unnecessary frequencies.

Ex: Frequencies which are harder to memorize and unnecessary

8:14 - 10:14, every 43 minute

The above frequency is not that human-friendly. The reason is that if a commuter wants to find the real departure time, then they need to add 43 minutes to starting time and calculate the correct departure time. This is an over work to do when that person has a busy schedule. The majority of the commuters are not willing to do this kind of calculation to simply find a departure time. For this reason, when generating a frequency map on public transit data, it is essential to only extract human-friendly frequencies and then include them in the frequency graph.

To achieve this goal, the frequency finding algorithm is modified as follows:

1. Start the algorithm with first uncovered departure time.
2. Check whether this departure time is the beginning of a frequency cover with one of the human friendly frequencies: 5 minutes, 7 minutes, 10 minutes, 15 minutes, 20 minutes, 30 minutes, 60 minutes, 120 minutes

To do this, check whether or not it is possible to build an arithmetic progression with any of these predefined human-friendly frequencies.

Ex: Let us consider the 5 minute frequency and assume initial departure time is  $t_0$ .

Then the algorithm checks whether there are departure times ( $t_1, t_2, t_3 \dots$  etc) as follows

---

**Algorithm 4** Algorithm for check availability of departure times

---

```
1 t1 = t0 + 5
2 t2 = t0 + 2*5
3 t3 = t0 + 3*5
4 tn = t0 + n * 5
```

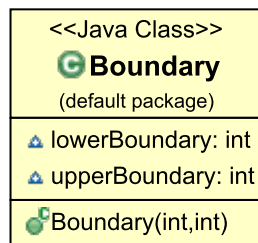
---

In this manner, the algorithm checks for all predefined human-friendly frequencies, to find the arithmetic progressions which are aligned with these frequencies. In real life, even though the streetcars have frequencies like 3 minutes, 5 minutes and 7 minutes, the railway network has frequencies like 15 minutes, 20 minutes, and 30 minutes. Therefore, for the Swiss railway network, we consider only the 1 hour and 2-hour frequencies, since 15, 20, and 30 minutes frequencies can be covered by one-hour frequency by generating more frequency lines between the starting and the destination node. One hour frequencies are drawn by normal lines while the two-hour frequencies are drawn by dashed lines.

The other issue with the above sample frequency is that the identified frequency is only valid for a very short period. It is again common to find that some public transit frequencies only exist during very early hours of the day or later hours.

1 Ex: 04:00 – 07:00, every one hour

These kind of frequencies are also unimportant when generating a frequency graph on public transit network because the frequency graph should consist of frequencies that at least last for the working hours/office hours of the weekdays. This is because these are the hours that most commuters use the public transit network and they are the main audience of the frequency map. To get rid of these kinds of unnecessary frequencies, the algorithm should check whether or not the identified frequency covered at least some specific times span of the day. This time span can be varied based on different factors such as connectivity of the transit network, weekday or a weekend, peak hours or off-peak hours etc., and for our implementation, we have used the time range of 10:30 to 15:30. This time range is defined using Boundary class, and it has two integer properties named lowerBoundary and upperBoundary.



**Figure 25:** Class diagram of Boundary class

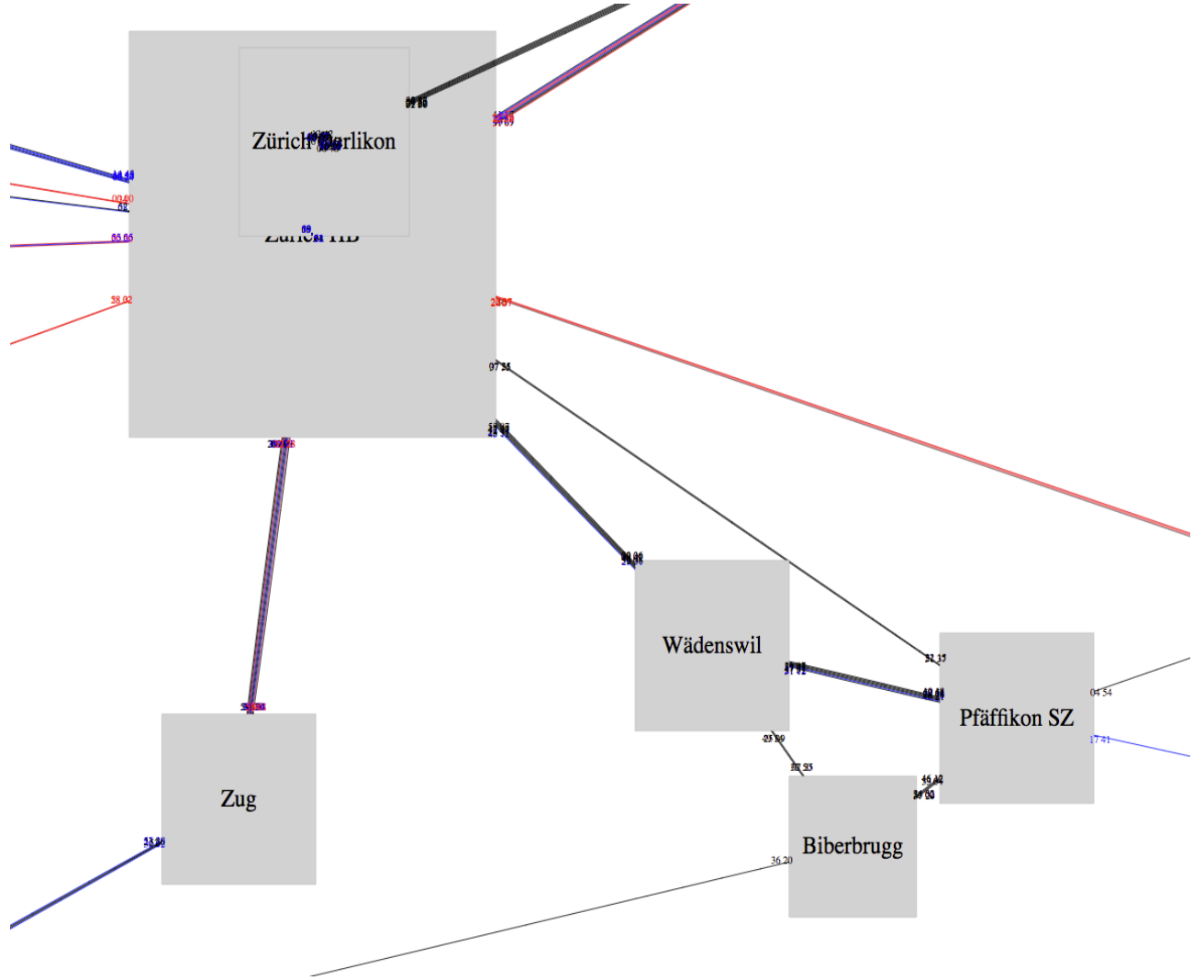
When a frequency is found, the algorithm checks if it satisfies this time boundary by checking if the initial departure time is less than or equal to the lower boundary and the final departure time is greater than or equal to upper boundary. If that condition is satisfied, then the algorithm categorizes the identified frequency as an eligible frequency for the frequency map.

### **4.3 Extraction of the frequencies among interesting nodes and creation of frequency lines.**

The Swiss railway network consists of around 8000 nodes/ stations. Finding the frequency between each of these 8000 nodes is unnecessary when generating a frequency map for the Swiss railway network. This is because when a frequency map is generated on a country level, it is sufficient to find the frequencies between selected nodes. The initial approach for selecting these nodes was to consider the number of in-connections and select the nodes where the number of in-connections exceeds a certain threshold. However, we found that this approach was ineffective. The reason for that is that most of the nodes around the major nodes like Zurich HB also get the same number of in connections since these nodes are intermediate nodes along the way to Zurich HB. When drawing a frequency map on a country level, it is not that useful to draw a frequency graph with nodes which located very close to each other. Due to this reason, in the final map, they are hard to differentiate from one another and drawing the frequency lines between them becomes useless due to the lack of readability.

A technical problem also arises when drawing the nodes and the frequency line between closely located nodes. The issue when generating a map on a country scale, that it is hard to draw two nodes which lie very close to each other, without overlapping the node boundaries. When the node boundaries collide with each other, the graphing tool (Graphviz) that is adapted to draw the frequency map, cannot draw orthogonal edges between these two nodes and all the frequency lines automatically converted into straight lines which are scattered with each other.





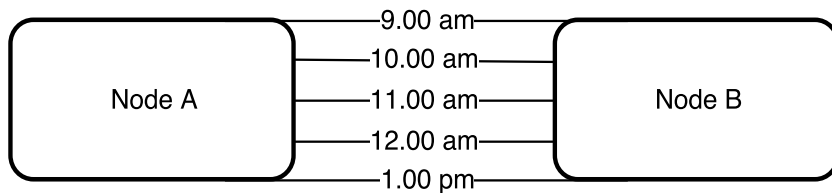
**Figure 26:** Example of nodes and edges overlapping and edges draw as straight lines

In the example above, the Zurich HB and the Zurich Oerlikon nodes are located very close to each other. When generating the frequency map, the approach of the program is to project the latitude and longitude values to a web mercator projection. (More details on this projection will be explained later in the implementation chapter.) After projecting coordinates for the node location, the size of the square shape which uses to represent the node are calculated based on the number of out-connections. The higher the number of out connections the larger the square that draws for the node. Therefore Zurich HB is drawn as a much larger square, and Zurich Oerlikon is drawn using a relatively small square based on the number of out-connection. This causes the graphing tool to draw the Zurich Oerlikon node inside the Zurich HB node. Once these nodes collide with each other, the Graphviz tool cannot render orthogonal lines between nodes and it causes graphviz tool to convert all the orthogonal lines

into straight line edges as it shown in the above diagram. When rendering all the lines as straight lines, the readability and the clarity of the frequency map greatly decreases, and subsequently as the usefulness of the map.

Therefore, the solution that identified to solve this problem is, the user of the program should feed the interesting nodes as a list, where the frequency map should be generated. In that case, the end user of the program has the complete control over which nodes should be considered and which nodes to eliminate in order to render a clear, consistent frequency map. If the frequency map is targeted only on the small geographical area then to overcome this problem it is possible to tweak the code responsible for generating the squares to represents the node. An even a better solution for this node overlapping problem is further described in the future work section.

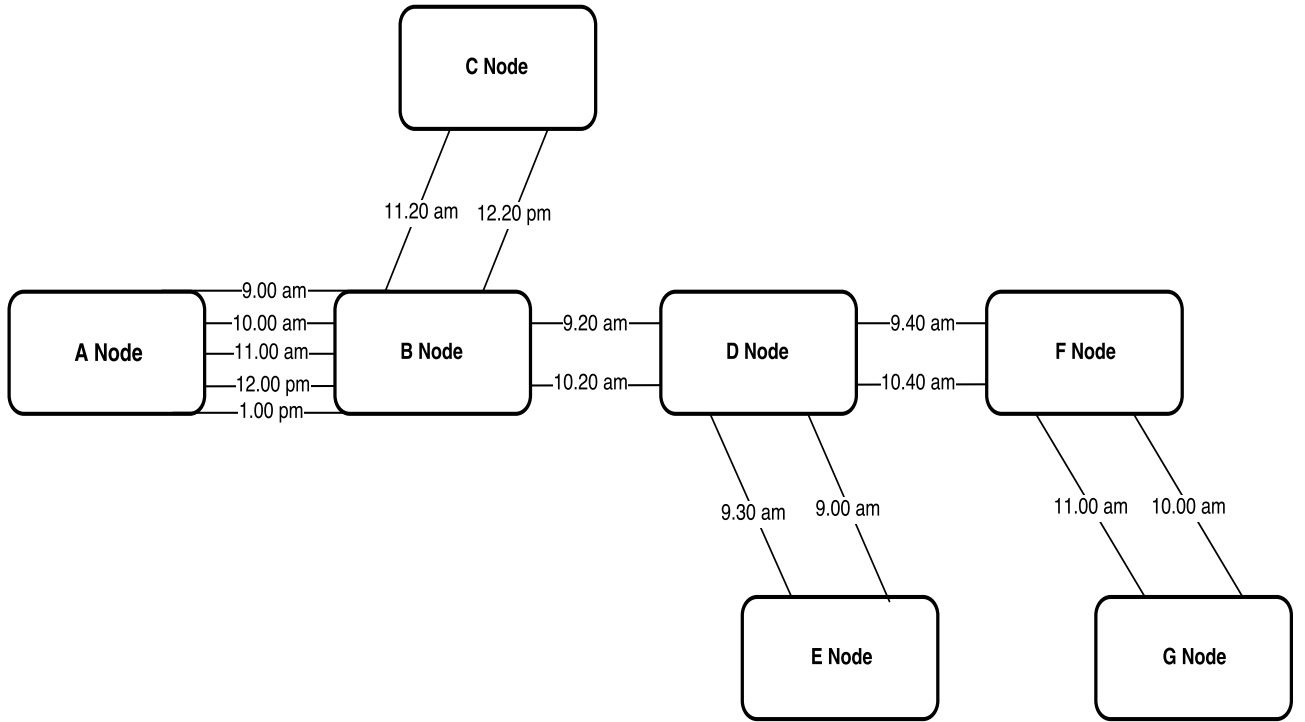
Even though the problem of selecting the nodes for the frequency maps is solved the initial problem of extracting the frequencies between these nodes is not solved yet. It is very easy to find the frequency covers of departure times between two consecutive nodes A and B. Since if the nodes are consecutive nodes; then there are out connections from node A and these out connections become in connections for the node B.



**Figure 27:** The diagram with two consecutive nodes named A and B and out connections from Node A to Node B

Therefore, in order to find the frequency cover from Node A to B, the only thing to do is to list all the departure times from A to B in ascending order and then pass it to the frequency finding algorithm.

The real problem comes when trying to find the frequency coverage between two distant nodes. Let us consider the situation of finding the frequency coverages between Node A and G.



**Figure 28:** The diagram with multiple nodes and out connections.

The frequency finding between these two distantly located nodes A and G is not as straightforward as finding the frequency between A and B.

One approach would be to navigate through all the out connections of node A and go to the next nodes, and then navigate through all the out connections of that node again and then keep progressing forward until finding the node G. This approach is similar to path finding from one node to the other node, but here we find the frequency covers between these two nodes, not the path. Therefore, to find the frequency covers, between A and G, it might need to run the algorithm again and again between all consecutive stations, and then extract the frequencies which are consistent among all these consecutive nodes.

Instead of above-mentioned approach, we have adapted another approach to finding the frequency between two distantly located nodes as per the following algorithm.

---

**Algorithm 5** Algorithm to find the frequency between two distantly located nodes

---

```
1 public ArrayList<String> searchTripsBetweenNodes(Node nodeA, Node nodeB) {  
2     Set<String> intersection = new HashSet<String>(nodeA.tripIDs);  
3     intersection.retainAll(nodeB.tripIDs);  
4     ArrayList<String> list = new ArrayList<String>();  
5  
6     for (String tripID : intersection) {  
7         if ((trips.get(tripID).nodeAndStopSequence.get(nodeB.id)  
8             - trips.get(tripID).nodeAndStopSequence.get(nodeA.id)) > 0) {  
9             list.add(tripID);  
10        }  
11    }  
12    return list;  
13 }
```

---

First, the algorithm gets the trip IDs of the trips which cover these nodes during the journey. These tripIDs are stored inside each node, while building the transit graph by reading the GTFS feeds. Then it gets the intersection of these two sets of trip IDs, and by now it has the trip IDs of all the trips which cover both of these nodes. Then it is necessary to check which direction the trip is following. In order to do that, the algorithm checks the stop sequence of each node inside the trip, and then gets the difference of the sequence and check whether it is positive or not. If it is positive, then this is a trip which covers both nodes and has the direction from node A to node B. In this manner the algorithm collects the common trip IDs which connects node A and B.

Then the `searchLeavingFrequencyFromStartStation()` method is getting called. This method is responsible for few major tasks in generating the frequency lines which are then used to draw the frequency map. Therefore it is convenient to analyze this method section by section.

---

**Algorithm 6** Algorithm for initiating of the frequency line generation: Part 01

---

```
1  ArrayList<FrequencyLine> frequencyLines;
2  String start_station_id;
3  String stop_station_id;
4  TripAndDuration initTrpAndDurItem;
5
6  private ArrayList<FrequencyLine> searchLeavingFrequencyFromStartStation(
7      String start_station_id, String stop_station_id,
8      ArrayList<String> commonTripIDs, LocalDate date) {
9      this.start_station_id = start_station_id;
10     this.stop_station_id = stop_station_id;
11     // To store the frequency lines.
12     frequencyLines = new ArrayList<FrequencyLine>();
13
14     Node start_station = nodeCollection.get(start_station_id);
15     // To store seconds in Integer
16     Set<Integer> timesInSeconds_set = new HashSet<Integer>();
17     ArrayList<Integer> timesInSeconds_list = new ArrayList<Integer>();
18
19     // To store the startingTimes and Trip IDS with duration.
20     TreeMap<String, ArrayList<TripAndDuration>> strtTimesWthTripID =
21         new TreeMap<String, ArrayList<TripAndDuration>>();
22
23     for (Edge outConnection : start_station.outConnections) {
24
25         if (commonTripIDs.contains(outConnection.trip_ID)
26             && isTripAvailable(outConnection.trip_ID, date)
27             && outConnection.start_time.getSeconds() >= 25200) {
28             timesInSeconds_set
29                 .add(Integer.valueOf(outConnection.start_time.getSeconds()));
30
31
32             if (!strtTimesWthTripID
33                 .containsKey(outConnection.start_time.getTimeInLongFormat())) {
34                 strtTimesWthTripID.put(
35                     outConnection.start_time.getTimeInLongFormat(),
36                     new ArrayList<TripAndDuration>());
37             }
38             ArrayList<TripAndDuration> trpIDAndDurList =
39                 strtTimesWthTripID
40                     .get(outConnection.start_time.getTimeInLongFormat());
41             String tripID = outConnection.trip_ID;
42             Time startTime = outConnection.start_time;
43             Trip trip = trips.get(tripID);
44             Time endTime = trip.nodeAndArrivalTimes.get(stop_station_id);
45             trpIDAndDurList
46                 .add(new TripAndDuration(outConnection.trip_ID, startTime, endTime));
47             strtTimesWthTripID.put(
48                 outConnection.start_time.getTimeInLongFormat(),
49                 trpIDAndDurList);
50         }
51     }
```

---

---

**Algorithm 7** Algorithm for initiating frequency line generation Part 02

---

```
1    timesInSeconds_list.addAll(timesInSeconds_set);
2    Collections.sort(timesInSeconds_list);
3    int[] secc =
4        timesInSeconds_list.stream().mapToInt(Integer::intValue).toArray();
5    Executor freqIdentifier = new Executor();
6    int minimumSequenceLength = 3;
7    int[] gapValues = {3600,7200};
8    // Frequency which covers 10:30:00am and 15:00:00.
9    Boundary boundary = new Boundary(37800, 54000);
10
11    freqIdentifier.execute(secc, minimumSequenceLength, gapValues, boundary);
12
13    for (int[] frqTimeSeries :freqIdentifier.finalSeriesInNumbers) {
14        if (frqTimeSeries.length > 1) {
15            List<Integer> finalSeriesIntegerList = Arrays.stream(frqTimeSeries)
16                .boxed().collect(Collectors.toList());
17            Time startTime = new Time(frqTimeSeries[0]);
18            ArrayList<TripAndDuration> tripDurListStartTime =
19                strtTimesWthTripID.get(startTime.getTimeInLongFormat());
20
21            for (TripAndDuration tripAndDuration :tripDurListStartTime) {
22                if (!tripAndDuration.considered) {
23                    ArrayList<TripAndDuration> selectedTrips =
24                        new ArrayList<TripAndDuration>();
25                    initTrpAndDurItem = tripAndDuration;
26                    selectedTrips.add(tripAndDuration);
27                    generateFrequencyLines(tripAndDuration, finalSeriesIntegerList,
28                        selectedTrips, strtTimesWthTripID);
29                }
30            }
31        }
32    }
33    return frequencyLines;
34 }
```

---

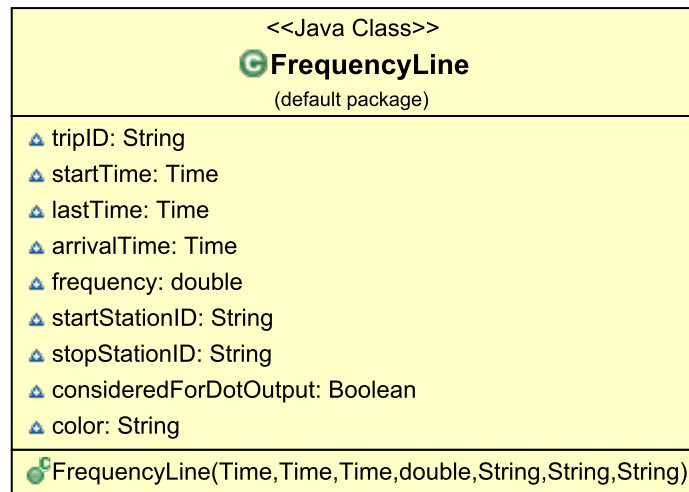
`searchLeavingFrequencyFromStartStation()` method accept four parameters.

- `start_station_id`: The node id of the start station
- `stop_station_id`: The node id of the destination station
- `common_trip_id`: The array list of trip ids which cover start and destination nodes
- `date`: The date which the frequency map should be drawn

The first step performed in this method is initializing the variables. This includes initialization of 03 local and 03 global variables which are used by the method.

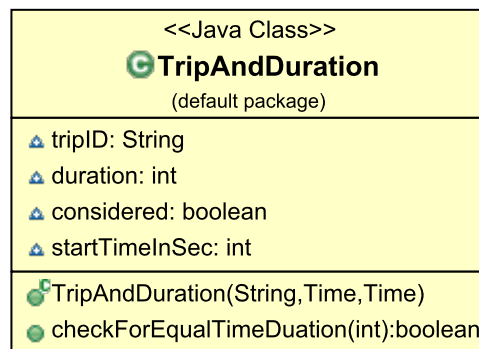
- `global start_station_id`: Make global reference in order to use `start_station_id` variable out of the method scope.
- `global stop_station_id`: Make global reference in order to use `stop_station_id` variable out of the method scope.
- `frequencyLines`: Arraylist of type `FrequencyLine`, to store the information relevant to frequency lines
- `timesInSeconds_set`: A set variable is used to eliminate the duplicate departure times from start station to destination station
- `timesInSeconds_list`: An array list variable is used to store the sorted departure times of `timesInSeconds_set`
- `strTimesWthTripID`: Map variable is used to store departure times as keys and `TripAndDuration` instances as values.

Frequency line is a model class that is used to store all the information related to a frequency line in the frequency map.



**Figure 29:** Class diagram of FrequencyLine class

TripAndDuration class is used to store the trip id and the time duration required to travel between the starting station and the destination station.



**Figure 30:** Class diagram of TripAndDuration class

When the frequency finding algorithm returns the frequency covers on departure times between the start station and the destination station, TripAndDuration instances are used to validate the frequency covers by analyzing the travel duration of each trip.

After initializing the properties, the algorithm then iterates over the out-connections of the starting node and checks for several conditions. First, the algorithm checks whether the trip id of the out connections is included in `common_trip_id` list. Secondly, the algorithm checks whether the trip is available on the given date and finally it checks for the starting time to be greater than 07:00 in order to exempt the unnecessary frequencies. If these conditions are met, then the departure time of that out connection is added to the `timesInSeconds_set`. Then an instance of TripAndDuration is created and added to the list which is mapped under



the key of the start time in the `strTimesWthTripID` map. At the end of this step, all the departure times of trips which travel from the starting node to the destination node are stored as keys in `strTimesWthTripID`. All the trips which start at a particular departure time are stored as `TripAndDuration` instances in a list and mapped with relevant departure time inside the `strTimesWthTripID` map. `TripAndDuration` instances consist of the trip id and the time taken to travel from starting node to the destination node.

As the next step, the algorithm inserts all the departure times of `timesInSeconds_set` into the `timesInSeconds_list` and sorts all the departure times in ascending order and assign all the numbers in the list to an int array. Then an instance of the frequency finding algorithm is created and we call the frequency finding method with the int array, `minimumSequenceLength`, `gapValues` and `boundary`. As the `minimumSequenceLength` the length of 03 is selected and, 1 hour and 2 hours are set as the `gapValues`. That means we only request the frequency covers which have 1 hour and 2-hour gap. The `boundary` is selected to filter the sequence which can cover time duration of 10:30:00 and 15:00:00 and beyond. Once these data are fed to the frequency finding algorithm, it returns all the arithmetic progressions it found as integer arrays.

The last step of the `searchLeavingFrequencyFromStartStation` method is to generate the frequency lines using the discovered frequency covers on departure times. The algorithm iterates through all the frequency covers resulted by the frequency finding algorithm and creates lists using these arrays. Now the problem is to correctly match the departure times which are returned by the frequency finding algorithm and the departure times which are stored at `strTimesWthTripID` map and finally to extract the correct trip ids for the departure times. This step is important because it is possible to have two or more trips which start from the starting node at the same departure time but with different journey durations. Therefore extracting the correct trip id from `strTimesWthTripID` for each departure time in frequency cover and validating whether each trip in the frequency cover has the same time duration as other trips is critical for generating good frequency lines.

In order to achieve this, a recursive method called `generateFrequencyLines()` is used and this method is initially getting called by `searchLeavingFrequencyFromStartStation` method.

---

**Algorithm 8** Algorithm for frequency line generation: Part 01

---

```
1 private void generateFrequencyLines(TripAndDuration crntTrpIDDur,
2     List<Integer> frqTimeSeries, ArrayList<TripAndDuration> selectedTrips,
3     TreeMap<String, ArrayList<TripAndDuration>> strtTimesWthTripID) {
4     int indxOfCurrTimeElmnt = frqTimeSeries
5         .indexOf(new Integer(crntTrpIDDur.startTimeInSec));
6
7     if (indxOfCurrTimeElmnt + 1 < frqTimeSeries.size()) {
8
9         Integer nextTimeElemenetInSeconds =
10             frqTimeSeries.get(indxOfCurrTimeElmnt + 1);
11         Time nextTimeItem = new Time(nextTimeElemenetInSeconds);
12
13         ArrayList<TripAndDuration> tripDurListStartTime =
14             strtTimesWthTripID.get(nextTimeItem.getTimeInLongFormat());
15
16         for (TripAndDuration tripAndDuration : tripDurListStartTime) {
17             if (!tripAndDuration.considered) {
18
19                 if (indxOfCurrTimeElmnt + 1 == frqTimeSeries.size() - 1) {
20                     if (initTrpAndDurItem
21                         .checkForEqualTimeDuation(tripAndDuration.duration)) {
22                         selectedTrips.add(tripAndDuration);
23                     }
24                     if (selectedTrips.size() >= frqTimeSeries.size() * 0.8) {
25                         Time startTime = new Time(frqTimeSeries.get(0));
26                         Time secondTimeElement = new Time(frqTimeSeries.get(1));
27                         Time lastTime =
28                             new Time(frqTimeSeries.get(frqTimeSeries.size() - 1));
29                         double frequency =
30                             (secondTimeElement.getSeconds() - startTime.getSeconds()
31                                 ) / 60;
```

---

---

**Algorithm 9** Algorithm for frequency line generation: Part 02

---

```
1      String tripID = selectedTrips.get(0).tripID;
2      System.out.println("Starting Trip Id " + tripID);
3      Trip trip = trips.get(tripID);
4      Time arrivalTime = trip.nodeAndArrivalTimes.get(
          stop_station_id);
5      FrequencyLine frequencyLine =
6          new FrequencyLine(startTime, lastTime, arrivalTime,
              frequency,
7              start_station_id, stop_station_id, tripID);
8      frequencyLines.add(frequencyLine);
9
10     for (TripAndDuration tripAndDurationItem :selectedTrips) {
11         tripAndDurationItem.considered = true;
12     }
13 }
14 break;
15 } else {
16     if (initTrpAndDurItem
17         .checkForEqualTimeDuation(tripAndDuration.duration)) {
18         selectedTrips.add(tripAndDuration);
19     }
20     generateFrequencyLines(tripAndDuration, frqTimeSeries,
21         selectedTrips, strtTimesWthTripID);
22     break;
23 }
24 }
25 }
26 }
27 }
```

---

generateFrequencyLines accept 04 parameters.

1. crntTrpIDDur: Current TripAndDuration instance being considered
2. frqTimeSeries: Frequency cover returns by frequency finding algorithm
3. selectedTrips: List of type TripAndDuration, to hold the TripAndDuration instances which are with valid time durations
4. strtTimesWthTripID: Reference to strtTimesWthTripID map, which has departure time as keys and list of type TripAndDuration as the value.

generateFrequencyLines is recursively getting called for each departure time in frqTimeSeries

and checks which instances of TripAndDuration which are stored in the strTimesWthTripID map have the same journey duration. During the last iteration of the frqTimeSeries the algorithm then checks the percentage of the TripAndDuration instances which have the equal time duration. If the percentage is greater than or equal to 80% then the frequency line instances are generated. The reason for checking the percentage of TripAndDuration instances with equal time durations is that there are some trips in the frequency covers which take more time or less time than the normal time required for the journey and most of the theses trips take place either early hours of the day or later hours of the day.

At the end of this process, all the frequency lines between nodes are created, and the next step is drawing the frequency map.

## 4.4 Drawing of frequency lines and nodes in a way that resembles a schematic map

Once all the frequency lines are generated the next major task is the drawing of the frequency graph. Graph drawing is a research topic which has been discussed for a long time, and it is still one of the trending research topics. Our main goal is to automatically generate a frequency map using a given GTFS feed. We tried several approaches until we ended up with acceptable results and this section is dedicated to discussing all those approaches and the results we achieved.

### 4.4.1 First approach by creating a web application using Leaflet

Leaflet [23] is a javascript based open source map library that can be used to render maps in mobile as well as the web. Here, the architecture we adapted is client-server architecture. The server is completely implemented in Java, and the client-side web application is implemented using HTML and JavaScript. The leaflet is embedded into the web application to draw the nodes and edges which users can interact with. When the web application loads, it makes a get request to the server by requesting all the nodes and node related information. Then the server responds with a JSON object which contains all the details relevant to nodes, including all of its out connections.

```
1 {  
2   "id": "139",  
3   "name": "Damm",  
4   "lat": 47.217327,  
5   "lon": 8.805061,
```

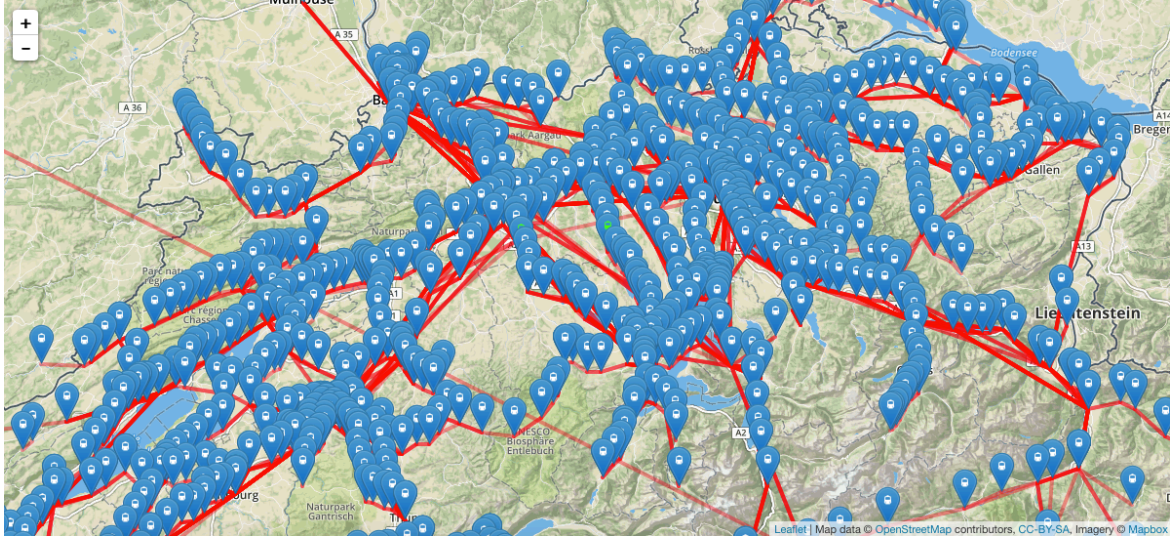
```

6    "timeZone": "null",
7    "outConnections": [{
8        "start_lat": 47.217327,
9        "start_lon": 8.805061,
10       "end_lat": 47.224888,
11       "end_lon": 8.81673,
12       "end_station_id": "8503110:3"
13   }, {
14       "start_lat": 47.217327,
15       "start_lon": 8.805061,
16       "end_lat": 47.214148,
17       "end_lon": 8.800729,
18       "end_station_id": "8503299:1"
19   }, {
20       "start_lat": 47.217327,
21       "start_lon": 8.805061,
22       "end_lat": 47.224888,
23       "end_lon": 8.81673,
24       "end_station_id": "8503110:2"
25   }, {
26       "start_lat": 47.217327,
27       "start_lon": 8.805061,
28       "end_lat": 47.202998,
29       "end_lon": 8.77811,
30       "end_station_id": "8503209:5"
31   }]
32 }

```

An example of a node described in JSON format and its out connections.

All the nodes are described by the server according to this JSON format, and when the JSON response is received by the web application, it extracts all the necessary information including latitude, longitude and its out connections before it renders the nodes and its edges in Leaflet map.



**Figure 31:** An image of Swiss railway network rendered by the Leaflet

This is a good example for reducing the readability of the map when the number of nodes and edges are unnecessarily high. In order to select two nodes to find the frequency cover, the map should be zoomed in to select the required node or otherwise it is highly possible to select an unnecessary node. Unnecessary crossings and overlapping of edges also cause degrade the quality of the map. One solution to this problem could be, rendering the nodes and edges only for the main nodes like Zurich HB, Basel, Olten etc. This will cause the removal of all the unnecessary nodes and edges from the map.

In order to find the frequency cover between two stations, two nodes should be selected in the map as well as the date from the calendar widget. The date is used to generate the frequency cover between the selected two nodes on that given date. Then the node ids of the selected nodes and the selected date are sent as a get request to the server, asking for the frequency cover between these two nodes. The server will run the frequency finding algorithm internally and returns the found frequency covers as a JSON response. Once the JSON response is received by the client-side web application, it extracts the information of frequency covers between selected nodes from the JSON response and presents it to the user.

Frequency Covers on: 2017-01-01

Time Series	Frequency Labels		
Time Series:(06:31:00, 07:01:00, 07:31:00, 08:01:00, 08:31:00, 09:01:00, 09:31:00, 10:01:00, 10:31:00, 11:01:00, 11:31:00, 12:01:00, 12:31:00, 13:01:00, 13:31:00, 14:01:00, 14:31:00, 15:01:00, 15:31:00, 16:01:00, 16:31:00, 17:01:00, 17:31:00, 18:01:00, 18:31:00, 19:01:00, 19:31:00, 20:01:00)	Start Time	End Time	Frequency
	06:31:00	20:01:00	30 minutes
Time Series:(21:01:00, 22:01:00, 23:01:00, 24:01:00)	Start Time	End Time	Frequency
	21:01:00	24:01:00	60 minutes

\*Individual time: time schedules which does not belong to any frequency label.

Close

**Figure 32:** Displaying the identified frequencies between Walterswil-Striegel and K ngoldingen

As we can see from the results, the approach works for finding the frequency cover between two nodes of interest. In practically, however, it is not acceptable since the commuters are highly unlikely to use a web application in order to find the frequency cover between two nodes. Instead of that, it is more convenient to use the timetable for this purpose.

After finding these drawbacks and challenges, we deviate from progressing on this approach and invest time on other alternative methodologies which can present the identified frequency covers and nodes in more convenient ways.

#### 4.4.2 Second approach using GeoJSON and QGIS

GeoJson is a specific JSON format which is used to describe geographical features. These features include polygons, lines strings etc. Our second approach is to describe the nodes as points, and frequency lines as line strings. Then we use an open source geographic information systems to render a frequency graph using this GeoJSON output.

In this approach conversion of standard JSON object which describes the nodes into a GeoJSON object was not a challenge. The only task here was to map the node related data into GeoJSON specific keys and output the JSON object. The following is a sample GeoJSON text which describes a node.

```

1 {
2   "type": "FeatureCollection",
3   "features": [{
4     "type": "Feature",
5     "geometry": {
6       "type": "Point",
7       "coordinates": [47.586826, 7.636695]
8     },
9     "properties": {
10      "name": "Weil am Rhein",
11      "id": "8014428"
12    }
13  }]
14 }

```

Explaining the frequency covers was not that easy using GeoJSON format. The reason for that is that in this approach only one line segment is drawn between any given two nodes A and B, in order to make the rendered map readable and clear. Therefore this line segment should consist with the frequency cover for both directions which are from A to B and B to A. This is achieved by describing the frequency covers using “from” and “to” keys and “+”, “-” signs. All this additional information is mentioned within the “properties” attribute in GeoJSON format. The following is an example for describing the frequency cover between two nodes using GeoJSON format.

```

1 {
2   "type": "FeatureCollection",
3   "features": [{
4     "type": "Feature",
5     "geometry": {
6       "type": "LineString",
7       "coordinates": [
8         [788044.2563034605, 5797324.319778668],
9         [819221.3947704418, 5816889.070265073]
10      ]
11    },
12    "properties": {
13      "from": "8501500",
14      "to": "8501506",
15      "frequencyCovers": [{

```

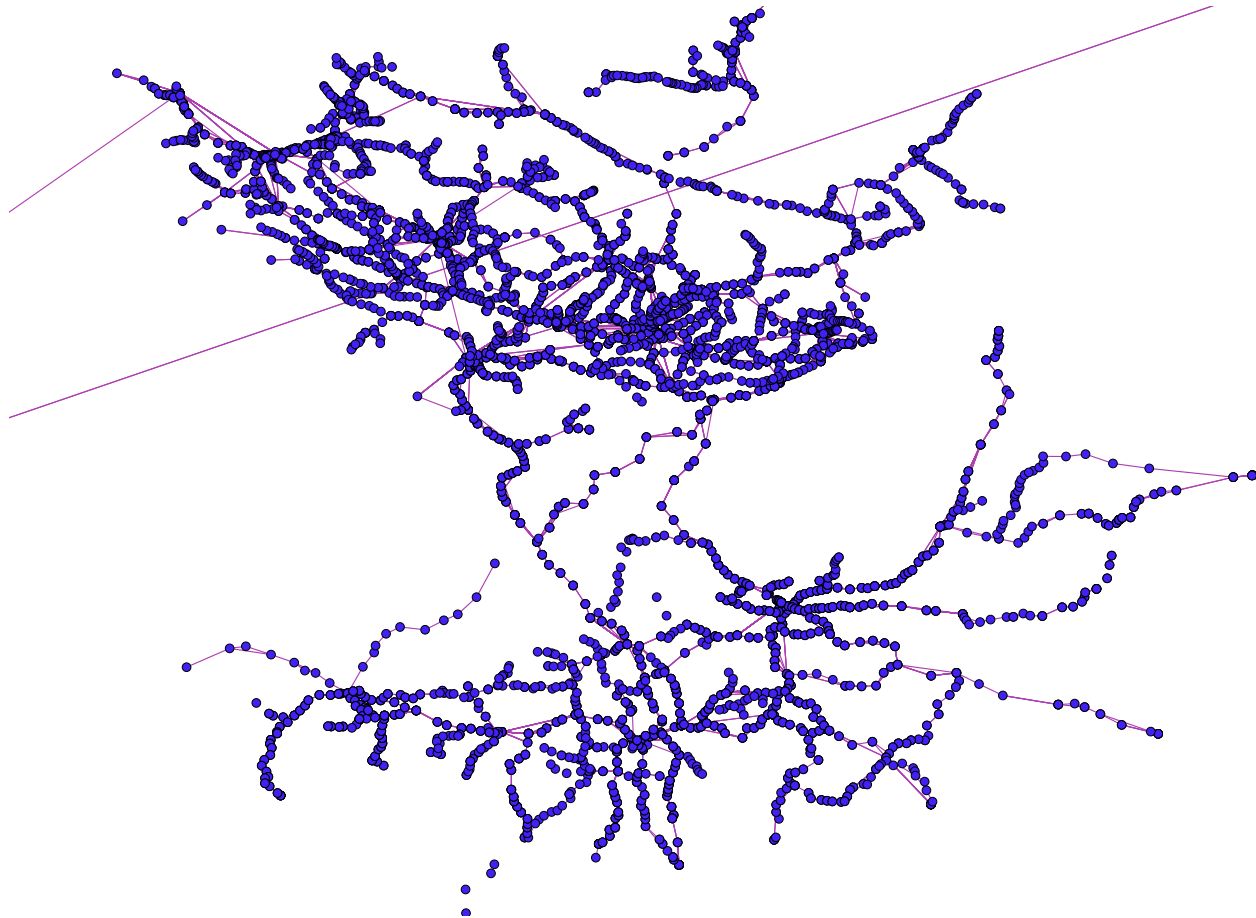


```

16     "frequency_covers": {
17     "time_frequencies": [{
18         "start_time": "07:43:00",
19         "end_time": "21:43:00",
20         "frequency": 60.0
21     }, {
22         "start_time": "06:09:00",
23         "end_time": "16:09:00",
24         "frequency": 120.0
25     }],
26     "individual_times": ["06:40:00", "07:11:00", "09:11:00", "11:09:00",
27         "13:09:00", "15:11:00", "17:09:00", "18:04:00", "18:17:00", "19:0
28         4:00", "19:17:00", "20:13:00", "21:09:00", "22:09:00", "23:15:00",
29         "23:46:00", "24:15:00", "25:20:00"]
30 },
31     "direction": "+"
32 }, {
33     "frequency_covers": {
34     "time_frequencies": [{
35         "start_time": "08:03:00",
36         "end_time": "20:03:00",
37         "frequency": 60.0
38     }, {
39         "start_time": "07:37:00",
40         "end_time": "17:37:00",
41         "frequency": 120.0
42     }],
43     "individual_times": ["04:29:00", "05:01:00", "05:33:00", "05:53:00",
44         "06:07:00", "06:31:00", "06:39:00", "07:07:00", "08:37:00", "10:3
45         5:00", "12:37:00", "14:35:00", "16:37:00", "18:37:00", "19:35:00",
46         "20:37:00", "21:01:00", "21:35:00", "22:03:00", "23:06:00"]
47 },
48     "direction": "-"
49 }
50 ]
51 ]
52 }

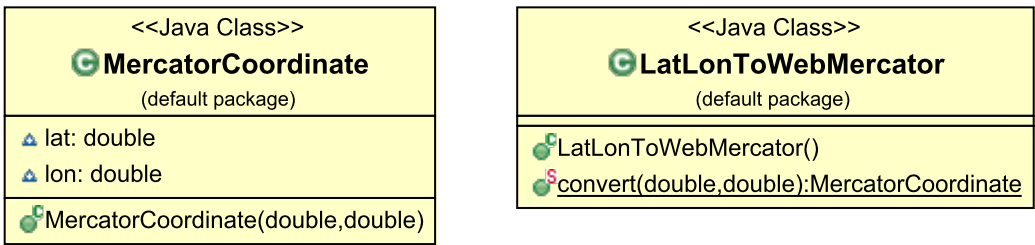
```

After generating the GeoJSON object consisting of all of the nodes and the edges, an open source GIS tool called QGIS [24] is used to render the map.



**Figure 33:** The initial version of the Switzerland railway map rendered by QGIS

The orientation of the initial map which is rendered by QGIS is not correct. The reason for that is QGIS required web mercator projection for the coordinates of the nodes and the edges. To achieve this, a custom class called LatLonToWebMercator is created for converting the latitude and longitude coordinates of nodes and edges into web mercator coordinate system.



**Figure 34:** Class diagram of LatLonToWebMercator class

---

**Algorithm 10** Algorithm for convert the latitude longitude values to web mercator projection coordinates.

---

```
1 public static MercatorCoordinate convert(double lat, double lon){
2
3     double x = 6378137.0 * lon * 0.017453292519943295;
4     double a = lat * 0.017453292519943295;
5     lon = x;
6     lat = 3189068.5 * Math.log((1.0 + Math.sin(a)) / (1.0 - Math.sin(a)));
7     MercatorCoordinate coordinate = new MercatorCoordinate(lat, lon);
8     return coordinate;
9 }
```

---

The function `convert()` mentioned above is used to convert the latitude longitude values to web mercator projection coordinates.

After converting all the latitude and longitude values to web mercator coordinates, the GeoJSON object is fed back to the QGIS and render the map again.



**Figure 35:** The Switzerland railway map rendered by QGIS with web mercator coordinates

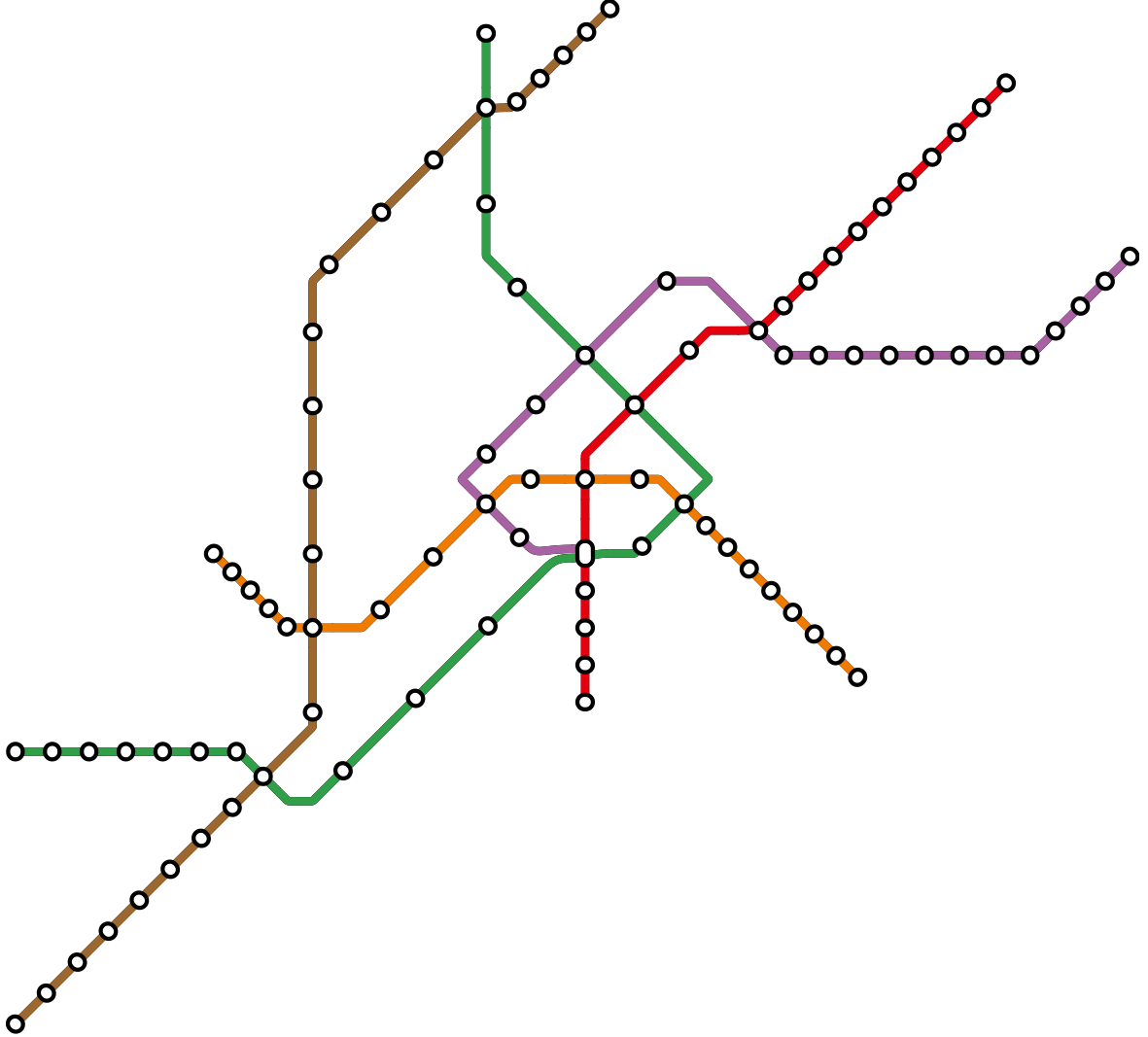
Now the map is much clearer and easier to read. Most of the nodes and the line segments are easily selectable compared with the web application. However, still, the details of the frequency covers are embedded into the line segment and lines should be selected to see the frequency cover information that is embedded in the line segment.

Selecting the edges to reveal the frequency cover is a bad user experience for the users of the frequency map. The next approach is to feed the GeoJSON file to the tool developed by chair for Data Structures And Algorithm [5] and generate a static frequency map using the tool.

#### 4.4.3 Third approach using GeoJSON and Octi

Octi is a tool developed under the chair for Algorithms and Data Structures [5], Faculty of Engineering of the University of Freiburg. It renders maps, using GeoJSON data, which describes the nodes and edges. Octi basically works by snapping station nodes to nodes on an

octilinear grid graph. it is a normal square grid graph [25] where every node is also connected by 45, 135, 225 and 315 degrees edges to its direct neighbours. It constructs the graph in a way that only allows shortest-path calculations between nodes on that graph that avoid acute angles to achieve smooth maps. This approach makes the map more readable and clean.



**Figure 36:** Subway network of Vienna rendered by Octi

Frequency graphs consist of many edges between nodes which represent the frequency coverage. However, Octi is still under active development and not yet able to render complex networks with many parallel lines due to the spacing issues. Therefore currently Octi is not capable of rendering frequency graphs, and maybe in future, it can be improved to draw frequency graphs conveniently.

#### 4.4.4 Fourth approach using Graphviz

Graphviz [15] is an open source graph drawing software which is used in many domains including software engineering, networking, machine learning etc. to draw diagrams. It consists of several software which can be used to draw and manipulate graphs [26]. Graphviz can draw diagrams by reading files which are written in Dot language. Dot is a graph description language which can describe graphs and their properties. Using Dot language, it is possible to define directed graphs [27], undirected graphs, and graphs with different node and edge styles etc. The code of the Graphviz tool can be found in Gitlab repository [28].

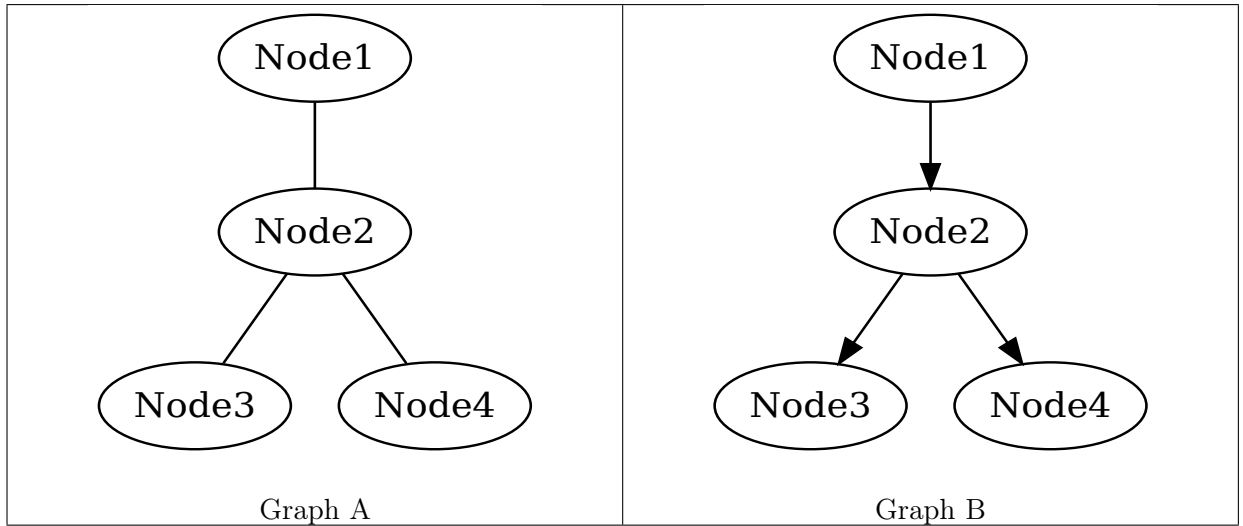
Let us consider two sample graphs written in dot language, in order to understand the basics of Dot language.

1	<code>graph g</code>	1	<code>digraph g</code>
2	<code>{</code>	2	<code>{</code>
3	<code>Node1 -- Node2 -- Node3;</code>	3	<code>Node1 -&gt; Node2 -&gt; Node3;</code>
4	<code>Node2 -- Node4;</code>	4	<code>Node2 -&gt; Node4;</code>
5	<code>}</code>	5	<code>}</code>
6	<code>Graph A</code>	6	<code>Graph B</code>

**Table 1:** Two sample graphs described in Dot language

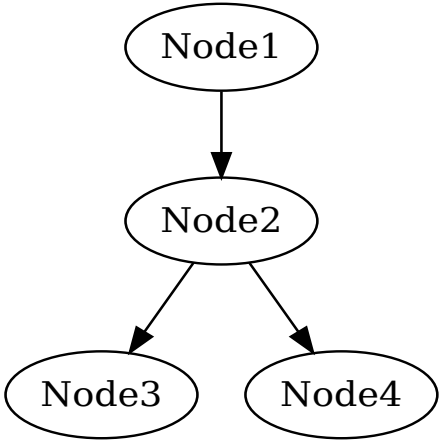
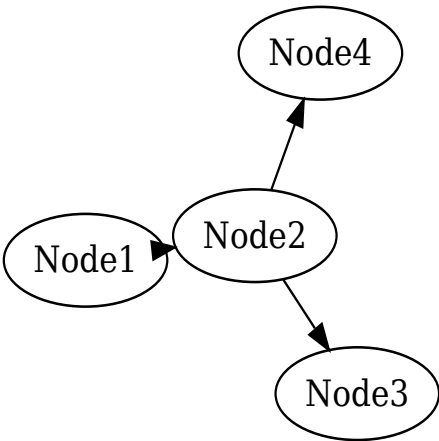
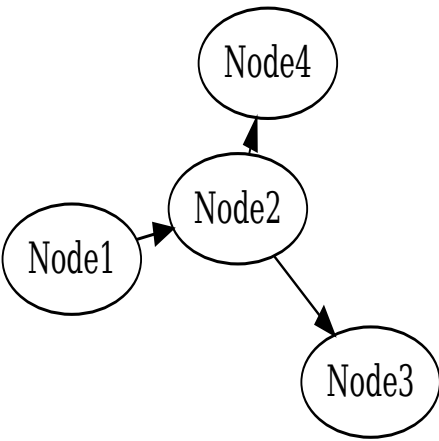
Graph A is an undirected graph with 4 nodes. “Node1 – Node2” means, there is an undirected edge between Node 1 and Node2. Graph B is a directed graph again with 04 nodes. “Node1 -> Node2” means the edge between Node 1 and Node 2 is directed and the edge is directed from Node1 to Node2. Basically, both graphs consist of four nodes and three edges and the only difference is Graph A consists of undirected edges and Graph B consists of directed edges.

These graphs are stored in files with “.dot” extension. Graphviz tool can read these dot files and draw graphs. Following are the rendered graph by Graphviz.

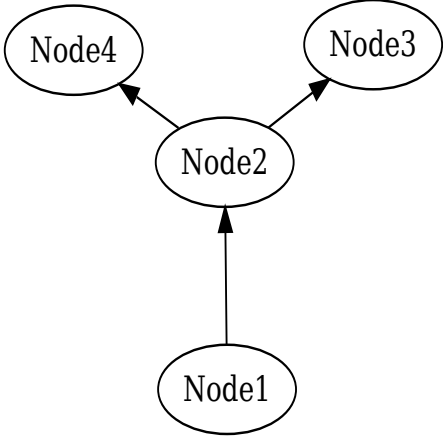
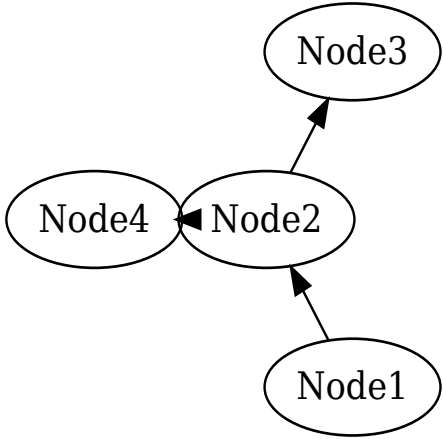
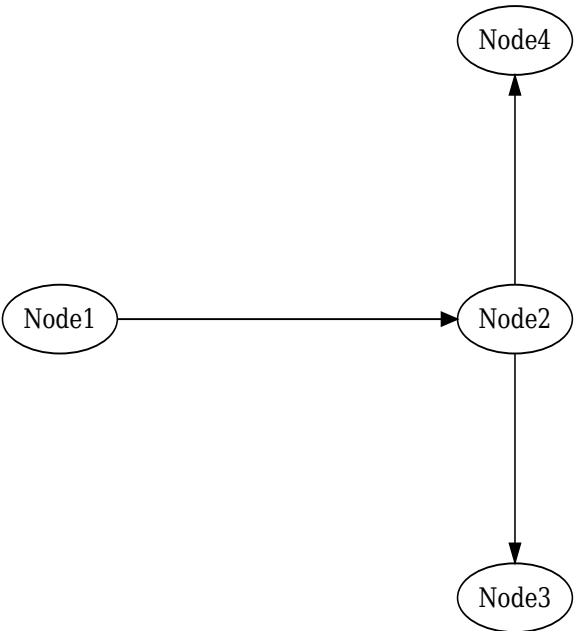


**Table 2:** Two sample graphs rendered by Graphviz

Graphviz has several layout engines that can be used to render graphs in dot language. According to the Graphviz official web site [16], the purpose for each layout engine is mentioned in the table below. The rendered graphs are also included in order to understand the type of graph which each layout can render.

<p><b>dot</b> - "hierarchical" or layered drawings of directed graphs. This is the default tool to use if edges have directionality.</p>	 <pre> graph TD     Node1([Node1]) --&gt; Node2([Node2])     Node2 --&gt; Node3([Node3])     Node2 --&gt; Node4([Node4]) </pre>
<p><b>neato</b> - "spring model" layouts. This is the default tool to use if the graph is not too large (about 100 nodes) and you don't know anything else about it. Neato attempts to minimize a global energy function, which is equivalent to statistical multi-dimensional scaling.</p>	 <pre> graph TD     Node1([Node1]) --&gt; Node2([Node2])     Node2 --&gt; Node3([Node3])     Node2 --&gt; Node4([Node4]) </pre>
<p><b>fdp</b> - "spring model" layouts similar to those of neato, but does this by reducing forces rather than working with energy.</p>	 <pre> graph TD     Node1([Node1]) --&gt; Node2([Node2])     Node2 --&gt; Node3([Node3])     Node2 --&gt; Node4([Node4]) </pre>



<p><b>sfdp</b> - multiscale version of fdp for the layout of large graphs.</p>	 <pre> graph BT     Node1((Node1)) --&gt; Node2((Node2))     Node2 --&gt; Node3((Node3))     Node2 --&gt; Node4((Node4)) </pre>
<p><b>twopi</b> -radial layouts, after Graham Wills 97. Nodes are placed on concentric circles depending their distance from a given root node.</p>	 <pre> graph BT     Node1((Node1)) --&gt; Node2((Node2))     Node2 --&gt; Node3((Node3))     Node2 --&gt; Node4((Node4)) </pre>
<p><b>circo</b> -circular layout, after Six and Tollis 99, Kauffman and Wiese 02. This is suitable for certain diagrams of multiple cyclic structures, such as certain telecommunications networks.</p>	 <pre> graph LR     Node1((Node1)) --&gt; Node2((Node2))     Node2 --&gt; Node3((Node3))     Node2 --&gt; Node4((Node4)) </pre>

After doing extensive experiments and evaluations, it was identified that the Neato layout engine is the best suitable candidate to draw frequency graph.

#### 4.4.4.1 Neato layout engine

According to the NetoGuide [29], “neato layout engine draws a graph by constructing a virtual physical model and running an iterative solver to find a low-energy configuration. Following an approach proposed by Kamada and Kawai [KK89], an ideal spring is placed between every pair of nodes such that its length is set to the shortest path distance between the endpoints. The springs push the nodes so their geometric distance in the layout approximates their path distance in the graph. This often yields reasonable layouts for the graph described in dot language”.

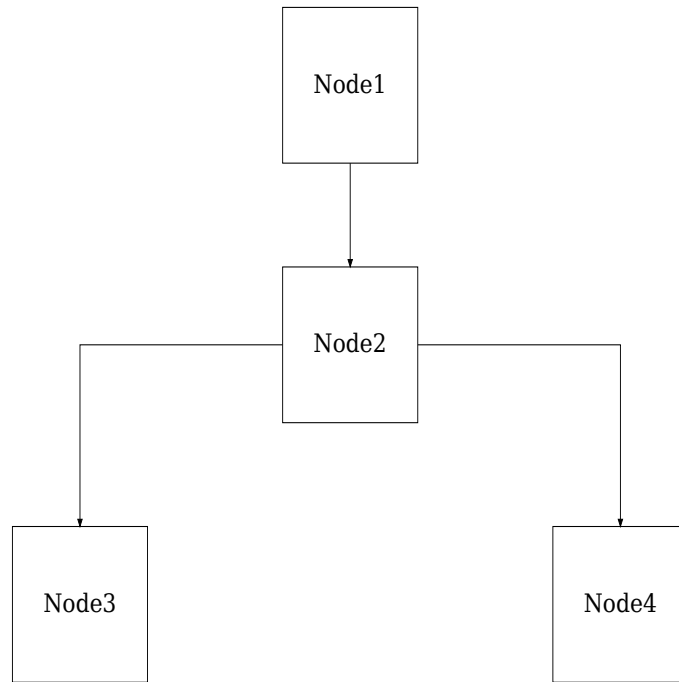
Another advantage in neato layout engine is that it is possible to give fixed positions for the nodes. This is important for our frequency graph since the location of nodes in the frequency graph should give a notion of information about the real geographical location of the stations. In order to make the edges between nodes readable and clear, an orthogonal edge style is used. An Orthogonal edge style makes the edges between nodes straight lines if possible or otherwise bend the edges an angle of 90 degrees when the edges turn up, down, left or right.

The following is a sample directed graph which is defined with fixed node positions, square node shapes, fixed node size, fixed font size and orthogonal edges.

```
1 digraph g {
2     splines=ortho;
3     Node1 [pos = "10,15!",fontsize=35, shape = box,width=3, height=3 ,label="
        Node1" ];
4     Node2 [pos = "10,10!",fontsize=35, shape = box,width=3, height=3 ,label="
        Node2" ];
5     Node3 [pos = "4,5!",fontsize=35, shape = box,width=3, height=3, label="
        Node3" ];
6     Node4 [pos = "16,5!",fontsize=35, shape = box,width=3, height=3 ,label="
        Node4" ];
7     Node1 -> Node2 -> Node3;
8     Node2 -> Node4;
9 }
```

Exclamation mark in the pos attribute of the node is used to instruct the Graphviz to use the fixed positions for the nodes as described in the dot file. The other attribute which is used above is self-explanatory attributes.

The rendered graph by Graphviz using neato layout engine for the above graph, is as follows.



**Figure 37:** Graph generated by neato layout engine using the orthogonal edge style and fixed node positions

According to the graph mentioned above, it seems neato layout engine is promising on drawing frequency maps with orthogonal edge routing. Therefore the found frequency covers according to section 3.2 as well as node details are described using dot language and created the GraphInDot.dot file. The selected data set is the 2017 Swiss railway GTFS data and the selected stations to draw the frequency map are as follows.

Stations that are selected to draw the frequency map Zurich HB, Olten, Basel SBB, Bern, Genève, Sargans, Chur, St. Gallen, Zug, Luzern, Pfäffikon SZ.

The following is the part of the created GraphInDot.dot file which describes the frequency map in dot language.

```

1 digraph G {
2     graph[size="70,20.25"]
3         splines=ortho;
4     edge[style=solid, penwidth="10",labeldistance="10"];
5     node [style="rounded,filled"];
6     8503000 [ pos = "2376.724561791971,8577.174708705374!", fontsize=300, shape
              = box, fixedsize=true, width=65, height=59, label= "Z{'u}rich HB" ];

```

```

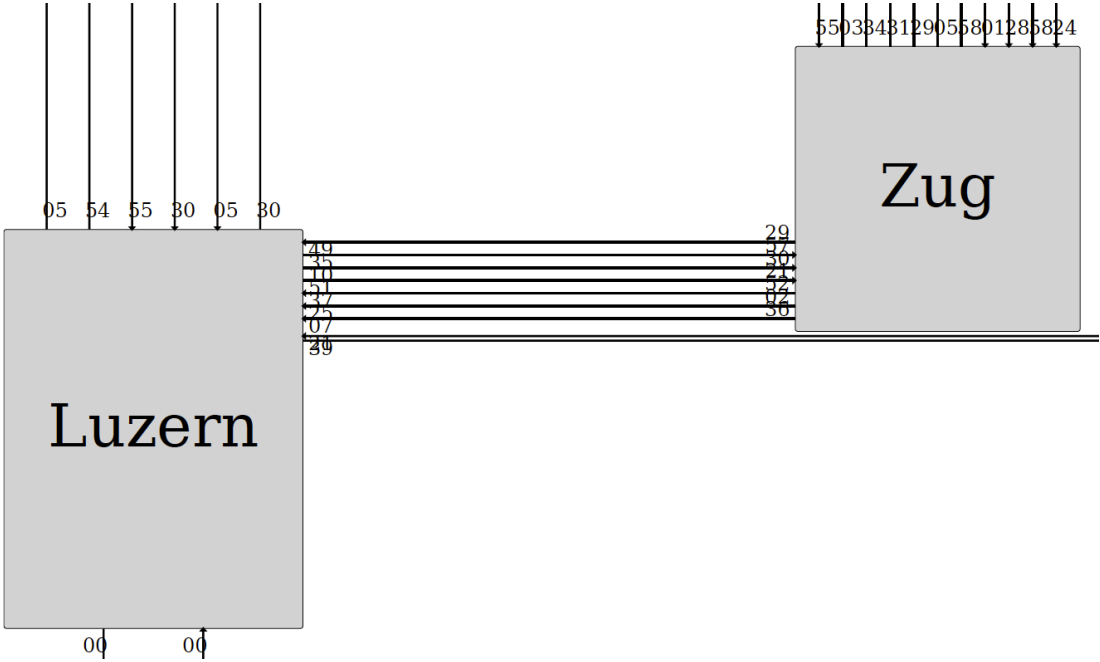
7      8500218 [ pos = "2200.6995037802,8571.011988937897!", fontsize=300, shape =
          box, fixedsize=true, width=20, height=20, label="Olten" ];
8
9      8503000 -> 8500218 [taillabel="03", headlabel="38" ,fontsize=100];
10
11     8503000 -> 8500218 [taillabel="06", headlabel="57" ,fontsize=100];
12
13     8503000 -> 8500218 [taillabel="55", headlabel="26" ,fontsize=100];
14
15     8503000 -> 8500218 [taillabel="30", headlabel="00" ,fontsize=100];
16     .
17     .
18     .
19 }

```

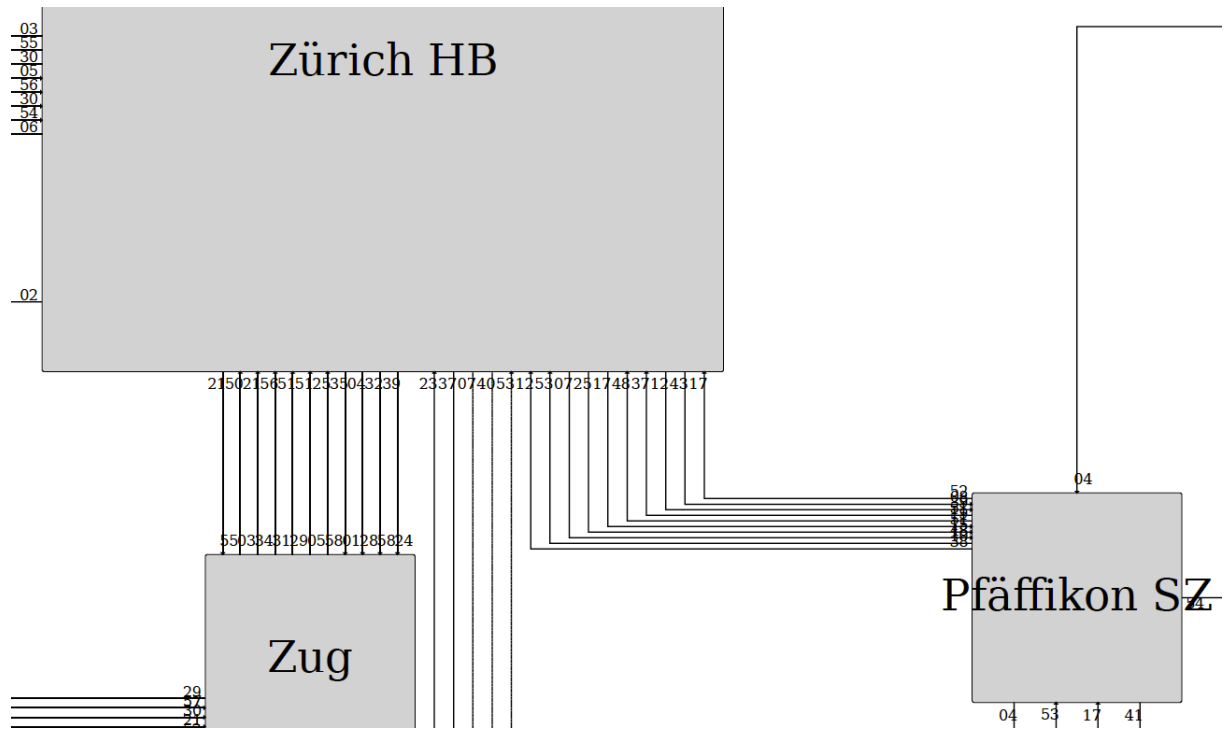
The GraphInDot.dot file then feeds in to the Graphviz program and renders a frequency map using neato layout engine.



In the rendered frequency graph above, all the frequency covers between the nodes are represented using directed orthogonal edges. Solid lines are used to represent the hourly frequency covers, and dashed lines are used to represent the two-hour frequency covers. Nearly 80% of the frequency graph is readable, but the frequency lines between some particular nodes are lying very close to each other, and sometimes they overlap with each other. This kind of output degrades the quality of the frequency map. An example of this scenario is frequency covers between Luzern and Zug stations as well as Zurich HB and Pfaffikon SZ stations.



**Figure 39:** Overlapping of parallel edges between Luzern and Zug



**Figure 40:** Overlapping of parallel edges between Züriich HB and Pfäffikon SZ

To overcome this edge overlapping problem, we have searched whether there are any solutions out there since Graphviz is a famous tool which is used by both academics and industry experts. We tried even the Graphviz gitlab page by opening an issue [30] for this problem. However, surprisingly there were not any solution out there, and we got 0 responses for the issue opened in gitlab. The reason for this edge overlapping will be explained in a later part of this chapter.

In the Switzerland Timetable-2017 [4] graph, the frequency lines consist of several colours which give the commuters the information about the type of the train which travels between nodes. Additionally, the Switzerland Timetable-2017 graph consists of bi-directional frequency edges, which helps to decrease the number of total frequency lines in the map and improve the readability.

#### 4.4.4.2 Prevent overlapping of parallel edges in Frequency Map

One major problem in the generated frequency map using Graphviz is that some edges/frequency lines between nodes are overlapping with each other or sometimes they lie too close to one another. Because of this, the frequency cover labels are lying on top of each other, and that makes the graph unusable for finding the frequency between those particular nodes. To

overcome this, two approaches were considered. The first approach is naive while the second approach is more standard.

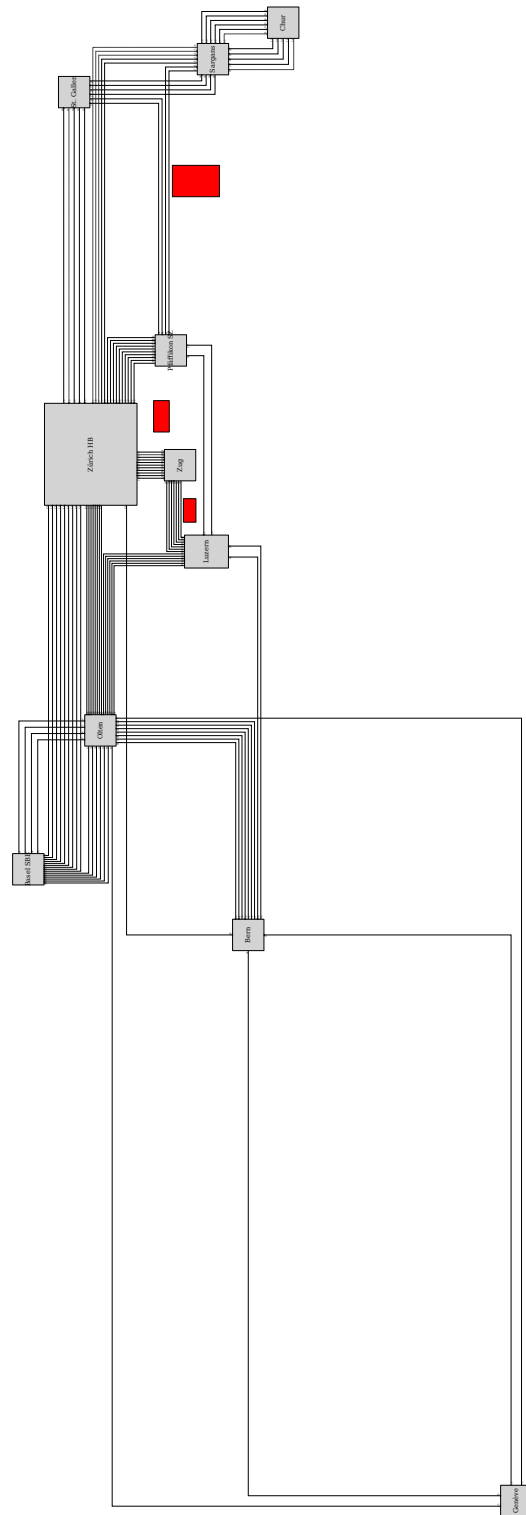
(a) **Using dummy nodes**

The first approach which was considered for overcoming edge overlapping problem is to introduce a few dummy nodes in the path where overlapping edges are routed. Then these dummy nodes act as obstacles for these overlapping edges and eventually, edges will be routed using some other paths and that might cause to remove the edge overlapping. To check the results of this approach three dummy nodes were introduced to the GraphInDot.dot file as follows.

```
1 digraph G {
2     .
3     .
4     .
5
6     "dummy1" [pos = "2400.9368383183564,8532.10287897316!", fontsize=300,
              fillcolor=red, shape = box, fixedsize=true, width=20, height=10,
              label="" ];
7     "dummy2" [pos = "2550.9368383183564,8510.10287897316!", fontsize=300,
              fillcolor=red, shape = box, fixedsize=true, width=20, height=30 ,
              label="" ];
8     "dummy3" [pos = "2340.9368383183564,8514.10287897316!", fontsize=300,
              fillcolor=red, shape = box, fixedsize=true, width=15, height=8 ,
              label="" ];
9     .
10    .
11    .
12 }
```

Dummy nodes which are dummy1, dummy2, dummy3 are included in the graph in a way that these dummy nodes will block the overlapping edges. Then the GraphInDot.dot file is fed to the Graphviz tool and rendered the frequency graph again.





**Figure 41:** Graph rendered after introducing dummy nodes

The nodes which are in red colour in the above graph are the dummy nodes that we introduced to the graph to block the overlapping edges. The introduction of the dummy nodes causes the overlapping edges to take alternative routes and that eventually removes the edge overlapping issue from the graph. Even though the result is accepted, the approach used here is completely naive. The introduction of dummy nodes to the graph is totally done in manually and this is an extra burden for the person who is going to use this tool for generating the frequency map. Even though the edge overlapping problem is solved here by adding dummy nodes, this is entirely a trial and error method. The introduction of dummy nodes to the path of overlapping edges may be removed the edge overlapping for that particular set of edges, but in the meantime it is possible that, the introduction of dummy nodes may also cause the edge overlapping in some other area of the map. Therefore this approach is not suitable for removing of edge overlapping in the frequency map.

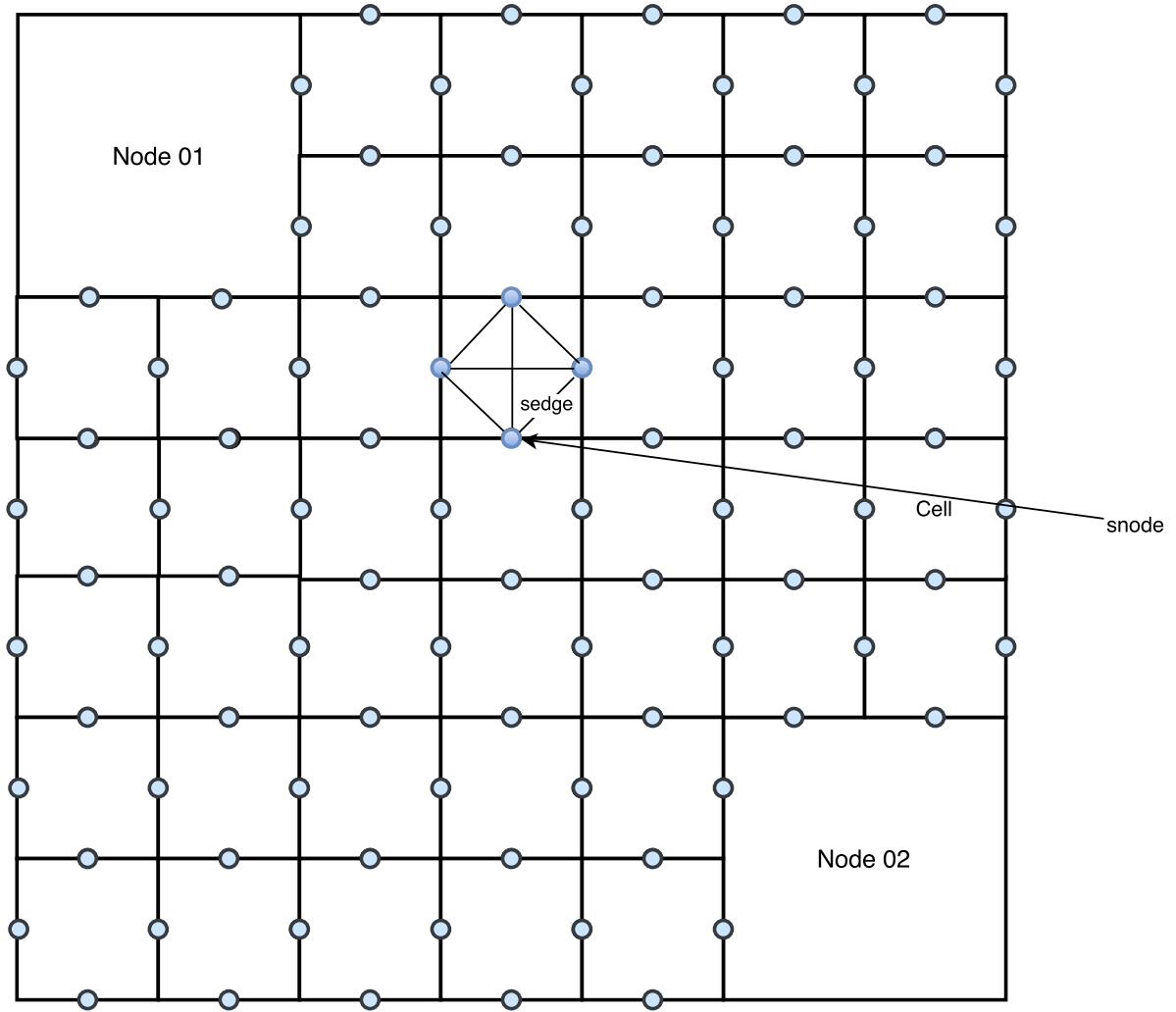
**(b) Fix the weight increase mechanism of edges.**

To explain this approach, it is essential to go through the fundamentals of the Orthogonal edge routing mechanism in Graphviz.

When a dot file is fed to the Graphviz tool, it first goes through the dot file and then creates a maze object with all the nodes in the dot file. This maze object consists of cells and nodes are also made of cells. Every cell has four sides and each side is consisting of snode. snode stands for search node. Within the cell, there are six edges which connect these snodes with each other, and these edges are called sedge. sedge stands for search edge. sedge have three attributes.

- weight: An integer variable which stores the weight of the edge. When frequency lines are routed, it always routes through the shortest path using edges with certain minimum weight.
- cnt: An integer variable which count the number of paths routed through this sedge.
- v1, v2: Two integer variables which hold the indexes of the snodes which sedge is connected with.

The following is a diagram which explains the structure and format of the maze, cells, snodes and sedges.

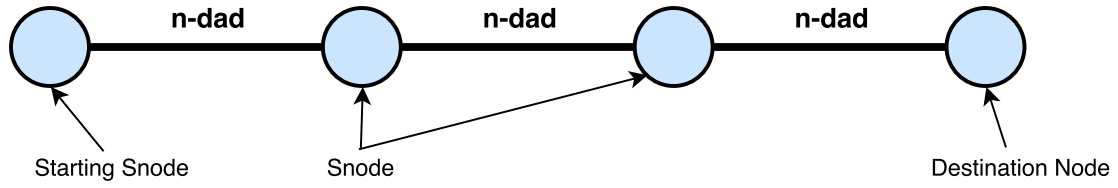


**Figure 42:** Maze consisting with cells

When a frequency line is drawing between nodes, it includes multiple steps. All these steps can be summarized as follows.

- First, the algorithm of Graphviz goes through all the nodes(nodes that are defined in dot file) and collects all the out edges into a collection.
- Then for each out edge, get the starting cell and destination cell and then create two snodes (sn and dn) which correspond to starting cell and destination cell.
- In the next step the graphviz algorithm finds the shortest path between sn and dn using the Dijkstra algorithm. When finding the shortest path, it navigates via snodes and its adjacent sedegs which has a weight which is less than some threshold value.

- d) Once the shortest path is found, the shortest path is stored by storing the reference to the next snode via **n\_dad** attribute of snodes.



**Figure 43:** The diagram with **n\_dad** between snodes

Then the algorithm updates the weight of the sedges which are part of the shortest path which links the snodes. To update the weights it uses `updateWt()` function.

- e) Once all the lines are routed as shortest paths, then the graph drawing starts and completes the drawing of the graph.

It is worth to note here that sedges are not real edges. These are virtual edges which define the connections between snodes. The lines are not directly drawn on top of sedges, instead they are routed between snodes of the sedges.

To increase the space between parallel edges, the mechanism of `updateWt()` function is changed. In the default implementation of the `updateWt()` function, it only updates the `cnt` variable of the sedge, when a line is routed through the sedge. However, this approach is modified as follows.

---

**Algorithm 11** Algorithm for increase the space between parallel edges

---

```
1 static void updateWt (cell* cp, sedge* ep, int sz)
2 {
3     ep->cnt++;
4     int x = 10;
5     double exponentValue = exp(((double)1/sz) * 200) ;
6
7     double alwdPaths = (double)sz / x;
8     double costForOnePath = ((double)BIG / alwdPaths) * exponentValue;
9     costForOnePath = (costForOnePath > BIG) ? BIG : costForOnePath;
10
11
12     ep->weight += costForOnePath;
13
14     // This was the default version of updateWt function
15     if (ep->cnt > sz) {
16         ep->cnt = 0;
17         ep->weight += BIG;
18     }
19 }
```

---

In the new version of the updateWt function, it calculates the amount of the weight attribute that should be increased using an exponential function. When the channel size is low, and a line is routed via a sedge, then the weight amount of the sedge increases by a large amount due to the exponential function which calculates the weight. If the channel size is high, the weight is increased by a lesser amount. By using this mechanism, the weight updating procedure is controlled. This new version of the updateWt function allows for more space between the parallel line which draws through the sedge by avoiding routing of a large number of lines through the sedge, which has low channel size. The increasing of the weight for sedges with smaller channel size causes the program to route the parallel lines using other sedges and snodes.

In the aforementioned method the value for 10 for x and multiplication factor for exponentValue of 200, are selected after experimenting with different values for several iterations. These values are subjected to the scaling parameters of the generated graph. The web mercator projection is adapted here to position the nodes inside the graph, and with the scaling of these coordinates, the above-selected values for x and the multiplication factor for exponentValue yield good results. The edges are comparably separated with each other, and the readability of the frequency labels are increased.

After fixing the spacing issue between parallel edges, the focus is then on introducing

colors for the frequency lines of the generated graph. This is achieved by checking the route type which is associated with trips.

---

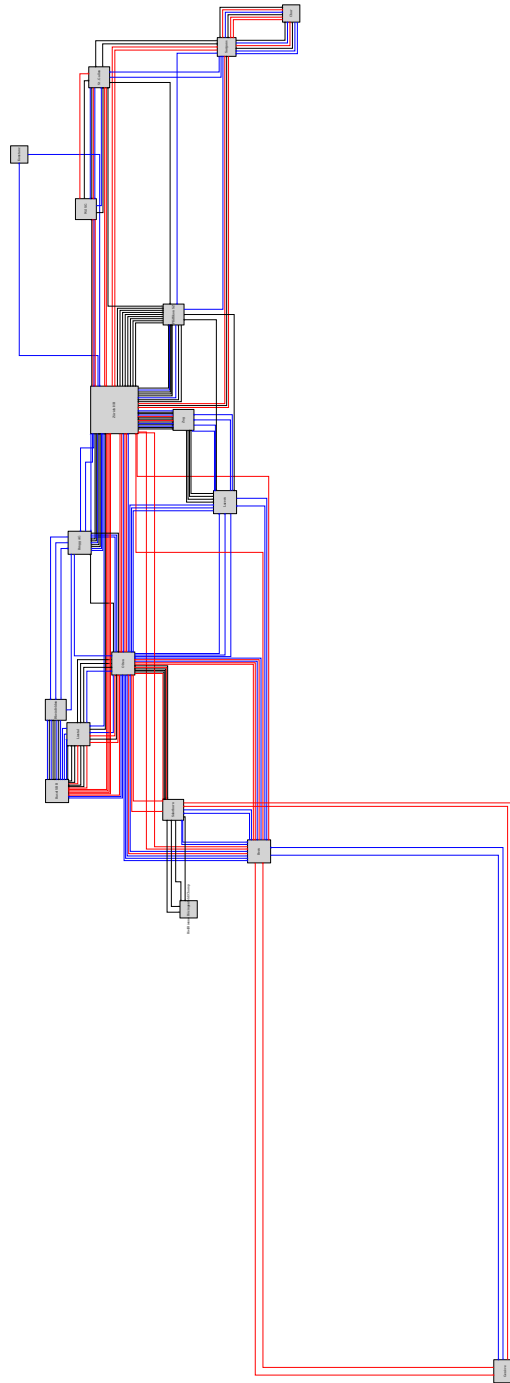
**Algorithm 12** Algorithm for introducing colors for the frequency lines

---

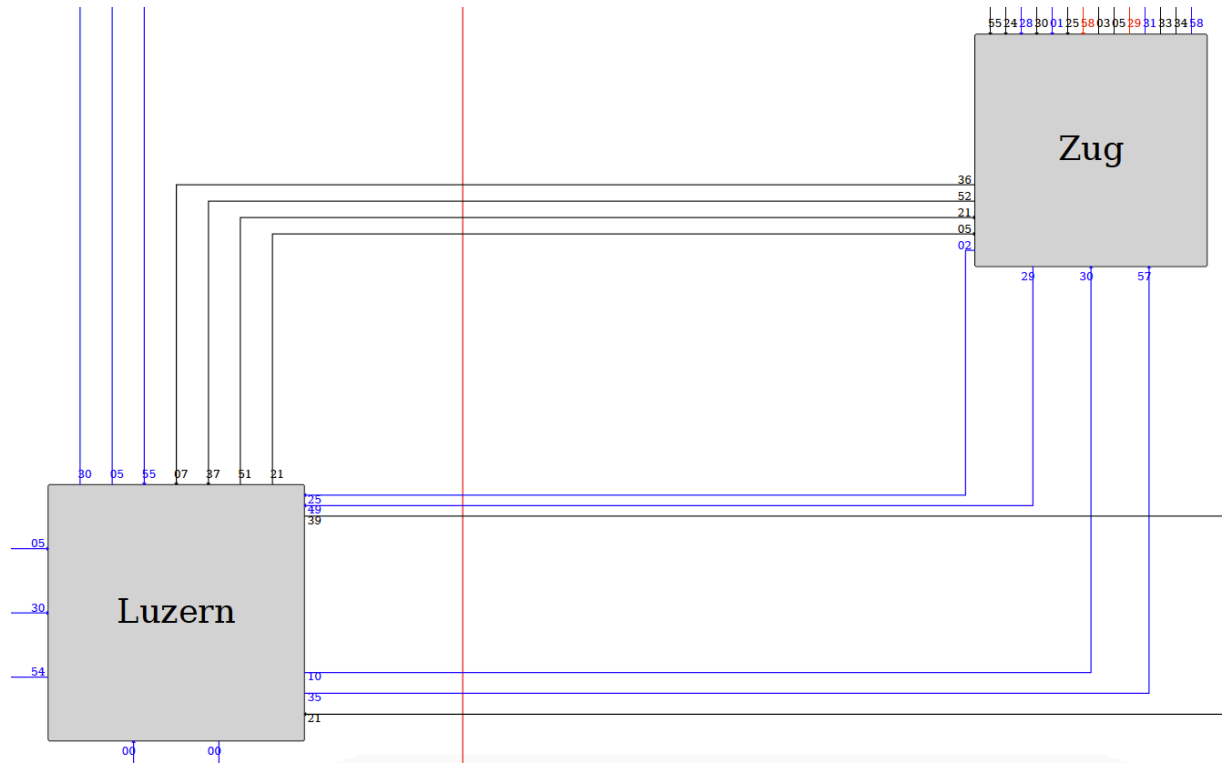
```
1 private String getEdgeAndFontColor(String routeType) {  
2     String color = "color=black, fontcolor=black";  
3  
4     if (routeType.contains("IC") || routeType.contains("ICE")  
5         || routeType.contains("TGV") || routeType.contains("RJ")) {  
6         color = "color=red, fontcolor=red";  
7     } else if (routeType.contains("IR") || routeType.contains("RE")) {  
8         color = "color=blue, fontcolor=blue";  
9     }  
10  
11     return color;  
12 }
```

---

Using the technique above, colours are introduced to the frequency lines, to make the frequency graph look similar to the Switzerland Timetable-2017 frequency graph. After all these modifications more nodes are added to the frequency graph in order to check the quality of the graph that is rendered by the Graphviz tool.



**Figure 44:** Graph generated with the new `updateWt()` function and the coloured frequency lines



**Figure 45:** Parallel edges between the Luzern and the Zug are not overlapping

Now the problem of overlapping edges is nearly solved. All of the frequency lines are readable and the edges are not overlapping with each other. However, because the frequency lines are unidirectional, the frequency map consists of a large number of frequency lines that represent all of the frequency coverages between nodes. This high number of frequency lines causes the frequency map looks messy and make it appear harder to access the frequency lines. The solution to this problem would be to merge all of the unidirectional frequency lines and make bidirectional frequency lines between nodes.

#### 4.4.4.3 Merging the unidirectional frequency lines and creation of the bidirectional frequency lines

1. Merge all the unidirectional frequency lines which are found in section 3.2 into bidirectional frequency lines.
2. Implement a mechanism in Graphviz to display two values in a head label and tail label.

Lets consider each step in a more detailed manner.



**Step 01:** The merging of unidirectional frequency lines and creation of the bidirectional frequency lines

Once all of the frequency lines are found according to the section 3.2 all of these frequency lines should be checked and processed.

---

**Algorithm 13** Algorithm for frequency lines checking and processing

---

```
1  ArrayList<FrequencyLine> frequencyLines =
2  searchFrequenciesOfTwoDistanceStations(nodeID1, nodeID2, date);
3
4  loop1: for (FrequencyLine frequencyLine :frequencyLines) {
5
6      String tripId = frequencyLine.tripID;
7      Trip trip = trips.get(tripId);
8      Route route = routes.get(trip.routeID);
9      TreeMap<String, Integer> stopsAndSequences = trip.nodeAndStopSequence;
10     for (String interstedNode :interestedNodes) {
11         if (stopsAndSequences.keySet().contains(interstedNode)
12             && interstedNode != nodeID1 && interstedNode != nodeID2
13             && (stopsAndSequences.get(interstedNode) > stopsAndSequences
14                 .get(nodeID1)
15                 && stopsAndSequences.get(interstedNode) < stopsAndSequences
16                 .get(nodeID2))) {
17             continue loop1;
18         }
19     }
20     frequencyLine.color = getEdgeAndFontColor(route.routeShortName);
21
22     String key =
23         frequencyLine.startStationID + "/" + frequencyLine.stopStationID;
24     if (allFrequencyLines.containsKey(key)) {
25         ArrayList<FrequencyLine> freqLineList = allFrequencyLines.get(key);
26         freqLineList.add(frequencyLine);
27         allFrequencyLines.put(key, freqLineList);
28     } else {
29         ArrayList<FrequencyLine> freqLineList =
30             new ArrayList<FrequencyLine>();
31         freqLineList.add(frequencyLine);
32         allFrequencyLines.put(key, freqLineList);
33     }
34
35 }
```

---

After finding the frequency coverages between two interesting nodes node1 and node2, the algorithm first checks whether any of the frequency coverage between node1 and node2 covers any of the interesting nodes. If a frequency coverage covers any of the interesting nodes, then that particular frequency coverage is ignored. The reason for this is, this frequency coverage will be discovered in future when finding the frequency coverage between node1 and the interesting node.

Ex: Let us assume that there are 03 nodes which are in a row  $A \rightarrow B \rightarrow C$ . If a frequency coverage between A and C also covers B, then that frequency coverage is ignored. Because this frequency coverage can be discovered when finding the frequency coverage between A and B and B and C.

After filtering the frequency coverages, the selected frequency coverages are stored in allFrequencyLines TreeMap. The key of the map is the node ids of node1 and node2 are separated by “/” character. The value for this key is a List object of type FrequencyLines and all selected frequency lines between node1 and node 2 will be stored inside this list collection. Once this step is completed for all the nodes, allFrequencyLines collection consists with all the frequency lines between all the interesting nodes.

The next step is the process of merging unidirectional frequency lines can be started.

---

**Algorithm 14** Algorithm for merging unidirectional frequency lines: Part 01

---

```
1  for (Entry<String, ArrayList<FrequencyLine>> entry :allFrequencyLines
2      .entrySet()) {
3      String keyComponents[] = entry.getKey().split("/");
4      String secondSetKey = keyComponents[1] + "/" + keyComponents[0];
5      ArrayList<FrequencyLine> frequencyLineSet1 = entry.getValue();
6      ArrayList<FrequencyLine> frequencyLineSet2 =
7          allFrequencyLines.get(secondSetKey);
8
9      for (FrequencyLine frequencyLine1 :frequencyLineSet1) {
10         if (!frequencyLine1.consideredForDotOutput) {
11             if (frequencyLineSet2 != null && !frequencyLineSet2.isEmpty()) {
12                 for (FrequencyLine frequencyLine2 :frequencyLineSet2) {
13                     if (!frequencyLine2.consideredForDotOutput
14                         && frequencyLine1.color.equals(frequencyLine2.color)
15                         && frequencyLine1.frequency == frequencyLine2.frequency) {
16                         graphInDot += "\n";
17                         graphInDot += frequencyLine1.startStationID + " -> "
18                             + frequencyLine1.stopStationID + "[taillabel=\""
19                             + frequencyLine2.arrivalTime.getTimeInLongFormat()
20                             .substring(3, 5)
21                             + " "
22                             + frequencyLine1.startTime.getTimeInLongFormat().
23                                 substring(3,
24                                     5)
25                             + "\"\" + \", headlabel=\""
26                             + frequencyLine2.startTime.getTimeInLongFormat().
27                                 substring(3,
28                                     5)
29                             + " "
30                             + frequencyLine1.arrivalTime.getTimeInLongFormat()
31                             .substring(3, 5)
32                             + "\"\" + \", \" + frequencyLine1.color + \" ,fontSize=130\"
33                             + \", arrowhead=none, arrowtail=none";
34                     if (frequencyLine1.frequency == 120) {
35                         graphInDot += ",style=\"dashed\"";
36                     }
37                     graphInDot += "]" + "; \n";
38
39                     frequencyLine1.consideredForDotOutput = true;
40                     frequencyLine2.consideredForDotOutput = true;
41                     break;
42                 }
43             }
44         }
45     }
```

---

---

**Algorithm 15** Algorithm for merging unidirectional frequency lines: Part 02

---

```
1      if (!frequencyLine1.consideredForDotOutput) {
2          graphInDot += "\n";
3          graphInDot +=
4              frequencyLine1.startStationID + " -> "
5                  + frequencyLine1.stopStationID + "[taillabel=\"
6                      + frequencyLine1.startTime.getTimeInLongFormat().
                          substring(3, 5)
7                          + \"\"\" + \", headlabel=\"\"\"
8                          + frequencyLine1.arrivalTime.getTimeInLongFormat().
                              substring(3,
9                                  5)
10                             + \"\"\" + \", \" + frequencyLine1.color + \" ,fontSize=130\"
11                             + \", arrowhead=normal, arrowtail=none\";
12          if (frequencyLine1.frequency == 120) {
13              graphInDot += \",style=\"dashed\"\";
14          }
15          graphInDot += "]" + "; \n";
16          frequencyLine1.consideredForDotOutput = true;
17      }
18  }
19 }
20 }
```

---

The code listed above in algorithm 14 will iterate through the allFrequencyLines collection and retrieve the Lists of type FrequencyLine. Then the algorithm checks for the colour and the frequency of the frequency lines which are in opposite directions between two nodes. If both the colour and the frequency are matching then, a bidirectional frequency line is created as a String object according to dot language standard. If a matching opposite directed frequency line cannot be found for a given frequency line, then in that case only, a uni directional frequency line is created. Through this method, all the unidirectional frequency lines are merged with the appropriate opposite directed frequency lines and the set of bidirectional frequency lines is created.

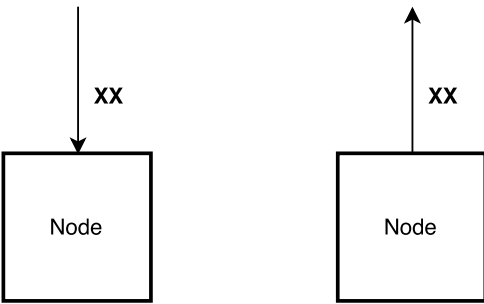
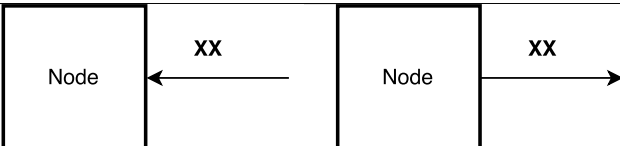
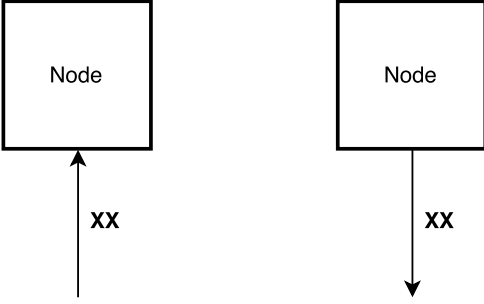
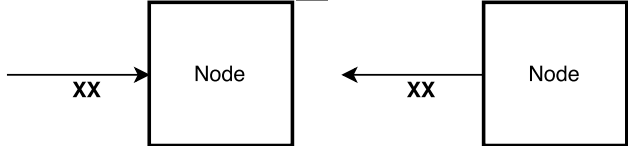
**Step 02:**Implement a mechanism in the Graphviz to display two values in the head label and tail label.

Graphviz only supports the single head and tail labels for edges. Therefore to render bidirectional frequency lines, a mechanism should be implemented within the Graphviz which enables the Graphviz to render bidirectional frequency lines successfully.

This is achieved by identifying the angle of the edge of which it is going to attach to the node

and then arranging the text inside the label.

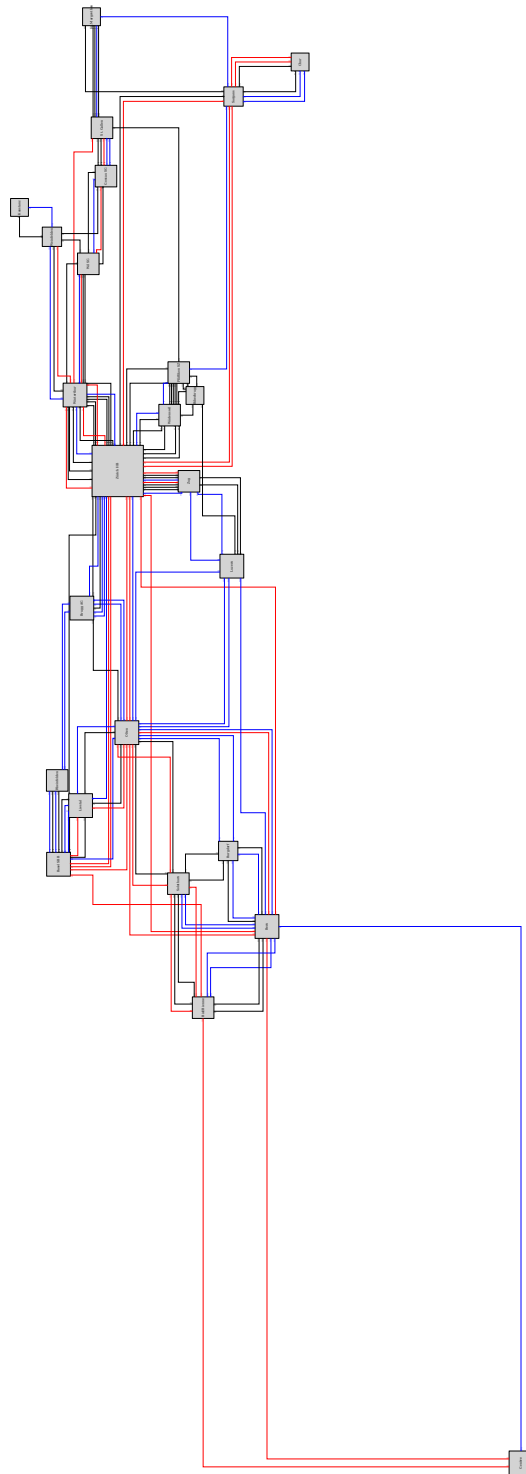
The following are the combinations of the angles of the edges when the edge is connected to a node.

Different ways of connecting orthogonal edges with node	Angle between orthogonal edge and the node
	1.13446
	-0.43633
	-2.00712
	2.70526

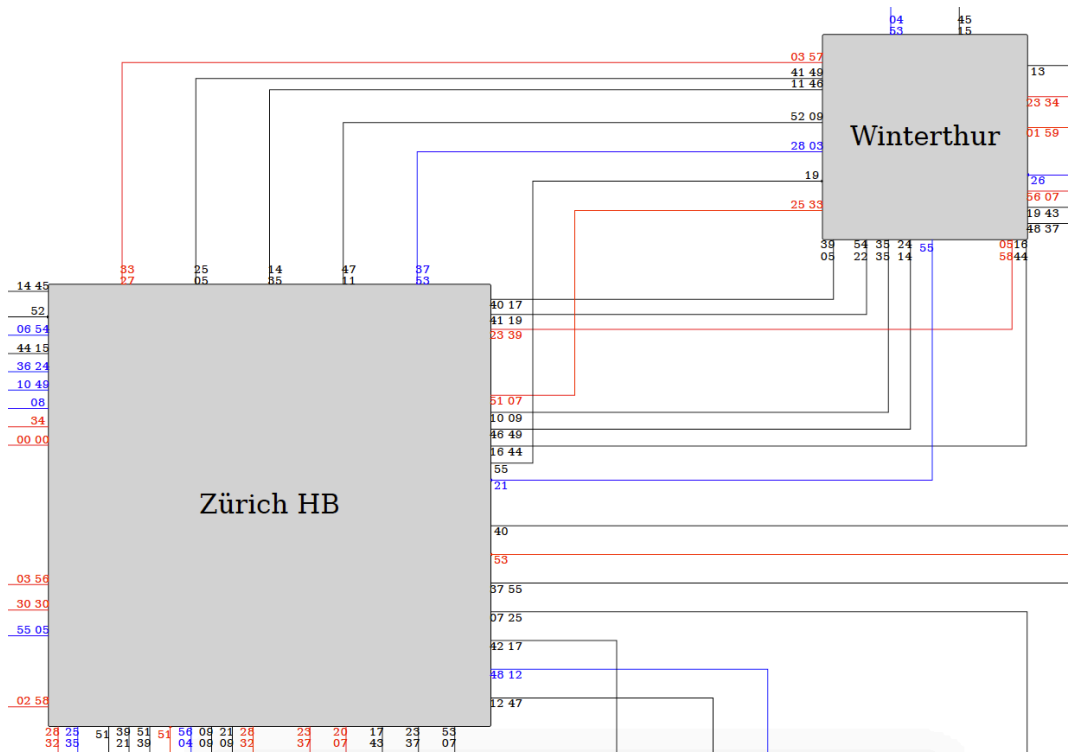
**Table 3:** This table shows different ways of connecting orthogonal edges with node and the angle between the node and the edges

After identifying the angle values successfully this information is then used to change the Graphviz code to give the feature of rendering two values in head and tail label of the edge.

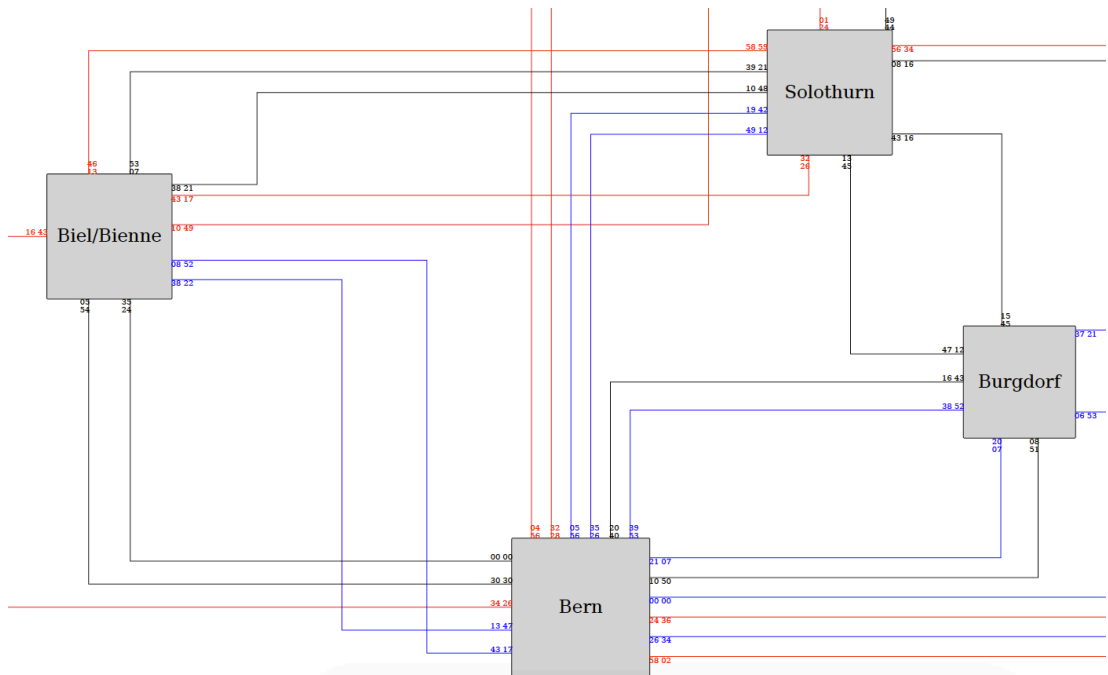
By changing the Graphviz code, the feature of rendering two values in head and tail labels of the edges of the map is implemented. Next, bidirectional frequency lines which are created by merging unidirectional frequency lines can be successfully drawn in the frequency map. Below is the graph with bidirectional frequency lines.



**Figure 46:** Frequency graph that generated with the bidirectional frequency lines



**Figure 47:** Bidirectional edges between Zürich HB and Winterthur



**Figure 48:** Bidirectional edges between few nodes



With bidirectional frequency lines, the graph is less clumsy and looks much cleaner. The end results are very much closer to the Switzerland Timetable-2017 frequency graph.

## 5 Evaluation

### 5.1 Evaluation of the time required to find frequency coverages using the frequency finding algorithm and the human readable frequency finding algorithm

To find the arithmetic progressions in a number sequence, two versions of the algorithms were developed. In the first version of the algorithm, it progresses through the number sequence by considering and summing up the gap between consecutive numbers. This version of the algorithm can find arithmetic progressions with any gap value, and it is called the frequency finding algorithm.

The second version of the algorithm finds the arithmetic progressions with predefined gap values, and this version is called the human-readable frequency finding algorithm. The rationale behind this is that normally public transit networks consist of frequency covers like 15 minutes, 20 minutes, 30 minutes, 1 hour and 2 hours etc. By considering this practical scenario, the second algorithm is developed in a way that it always find for the arithmetic progressions with predefined gap values as mentioned above. The advantage of this approach is to reduce the running time of the algorithm as well as getting rid of frequency coverages which are not human-friendly.

Ex: 9:00 - 19:00, every 17th minute

Frequencies as the above are not of much use, because then commuters need to add 17 minutes gap value consecutively to the starting time of the frequency cover in order to find the correct departure time for a journey.

To evaluate the running time between these two versions of the algorithm, the Swiss Railway GTFS dataset is used. The size of the data and the overall availability of good frequency coverages allows this data set, to be used as a base reference for the running time evaluation.

The frequency finding algorithm ran five times on the data set and recorded time duration that needed for the execution.

The configurations of the computer which the evaluation is conducted is as follows.

- Processor: Intel(R) Xeon(R) CPU E5640 @ 2.67GHz
- Memory: 65 GB

As the next step, the human friendly frequency finding algorithm ran on the same data set and recorded the time taken to find the frequency covers with predefined gap values.

Round	Time taken in milliseconds
1	11096
2	11095
3	11124
4	11051
5	11110

**Table 4:** Running time of frequency finding algorithm

Round	Time taken in milliseconds
1	9516
2	9636
3	9405
4	9243
5	9281

**Table 5:** Running time of human-friendly frequency finding algorithm

The average running time taken to find the frequency covers by the frequency finding algorithm is 11095 ms and the algorithm occupied 6 GB of memory.

With the predefined gap values (1 hour and 2 hour gaps), the human friendly frequency algorithm finds the frequency coverages in average of 9416 ms and the algorithm occupied 4.5 GB of memory.

As expected, the human friendly frequency finding algorithm executes faster than the normal frequency finding algorithm.

## 5.2 Evaluation for the time required to build the transit graph

The next step of the evaluation chapter is to evaluate the average running time that is required to build the transit graph on different GTFS data sets. This is done on two GTFS data feeds, specifically the Swiss Railway GTFS dataset and the Deutsche Bahn GTFS dataset.

For the Swiss Railway GTFS data set, the size of the data set as well as the average running time is as follows.

File	Size of the file	Number of records
calendar_dates.txt	148.8 MB	6304173
calendar.txt	127 bytes	1
routes.txt	1.6 MB	34100
stop_times.txt	68.9 MB	1512166
stops.txt	541 KB	8054
trips.txt	7.7 MB	127329

The algorithm for building the transit graph is run on this data set for several times and the average running time that required for build the transit graph is 75975 ms.

Then the transit graph building algorithm tested on the Deutsche Bahn data set and the size of the data set as well as the average running time as follows.

File	Size of the file	Number of records
calendar_dates.txt	504 KB	32125
calendar.txt	101 KB	2766
routes.txt	26 KB	1356
stop_times.txt	3.6 MB	115467
stops.txt	24 KB	595
trips.txt	229 KB	7818

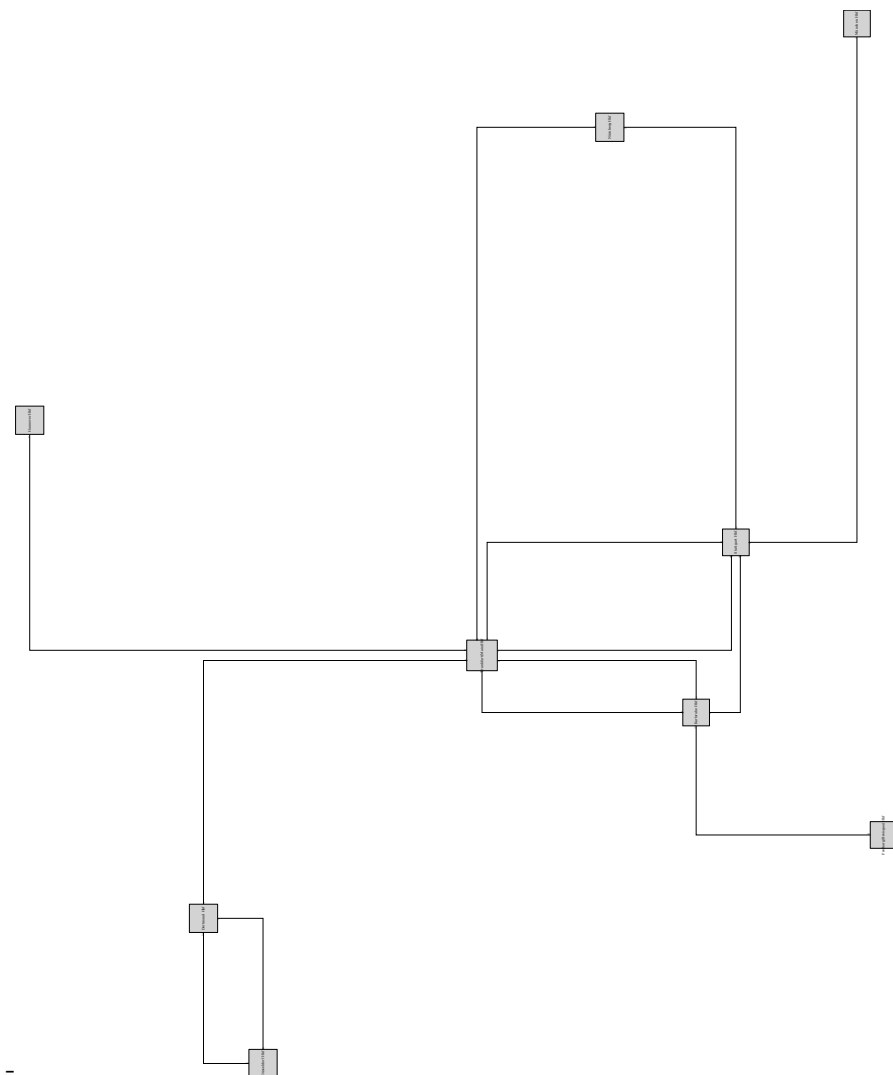
The algorithm required an average running time of 3956 ms to build the transit graph using the Deutsche Bahn dataset.

The Deutsche Bahn dataset is significantly smaller when compared to the Swiss Railway GTFS dataset, the reason for that is, the short distance trips are filtered out from the data set since short distance journeys do not make any contribution when finding the frequency covers between major cities. The reduced size of the data set results in a lower running time of the transit graph building algorithm.

### 5.3 Evaluation of the frequency graphs that were rendered on the Swiss Railway Network GTFS dataset and the Deutsche Bahn GTFS dataset

Frequency graphs are generated on two sets of GTFS feeds.

1. Swiss Railway GTFS feed
2. Deutsche Bahn GTFS feed



**Figure 49:** Frequency graph generated on Deutsche Bahn GTFS data



When compared to the Swiss dataset, the Deutsche Bahn dataset has a significantly lower number of frequency lines between the nodes. The reason for this is that the trips in the Deutsche Bahn transit network do not have high consistency and equal travel times. The consistency of departure times of the journeys in the Deutsche railway network slightly differs in few minutes. This slight difference makes the algorithm not able to identify the frequency covers in the Deutsche Bahn transit network.

An example of this scenario is when finding the departure times between the Freiburg HBF and Karlsruhe HBF the departure times from Freiburg HBF as follows.

8:57, 9:56, 10:57, 11:56, 12:57, 13:57 ..

Due to the difference of 1 minute in the departure times, the algorithm cannot identify this departure time sequence as a frequency line. However, this could be fixed and, the suggested solution will be further discussed in the future work chapter.

## 5.4 Evaluation on the trips covered by the frequency coverages out of the total trips

In order to measure the quality of the frequency graph, another dimension that we can use is to measure the total number of trips covered by the frequency graph. Every trip has its own departure time. Therefore the measuring of the number of trip coverage achieved by the frequency covers can be done by measuring the total number of departure times covered by the frequency covers. This is done for some selected nodes in both GTFS datasets.

Start and stop station	Total number of departure times	Total number of departure times covered by the frequency covers	Percentage
Zurich HBF - Olten	68	63	92.64%
Zurich HBF - Basel SBB	77	77	100%
Olten - Bern	84	76	90.47%
Geneve - Luzern	15	14	93.33%
Sargans - Chur	88	83	94.31%

**Table 6:** For the selected nodes, the total number of departure times covered by frequency coverages in the Swiss GTFS dataset

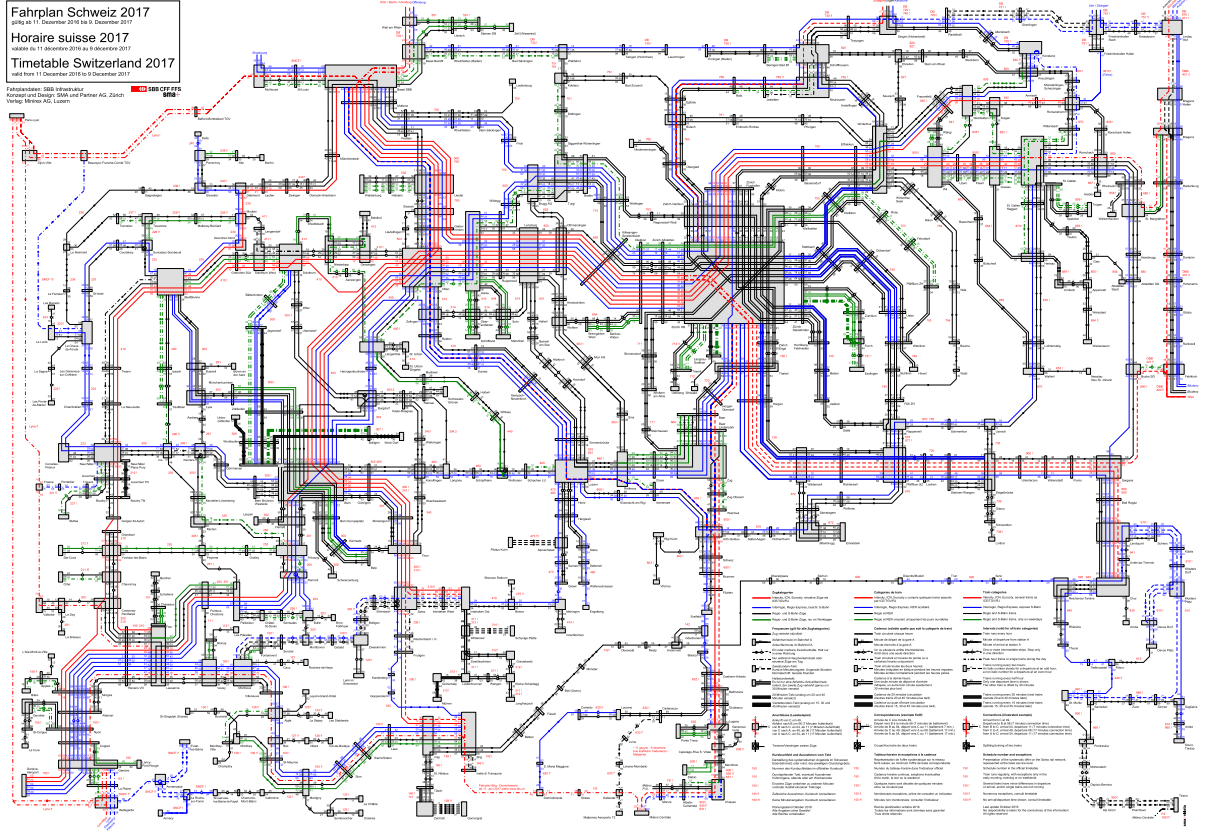
Start and stop station	Total number of departure times	Total number of departure times covered by the frequency covers	Percentage
Freiburg(Breisgau) Hbf - Frankfurt(Main)Hbf	12	6	50%
Karlsruhe Hbf - Stuttgart Hbf	20	8	40%
Frankfurt(Main)Hbf - Munchen Hbf	9	6	66.66%
Stuttgart Hbf - Nuremberg Hbf	7	7	100%
Dusseldorf HBF - Stuttgart HBF	11	0	0%

**Table 7:** For the selected nodes, the total numbers of departure times covered by the frequency coverages in the Deutsche Bahn GTFS data

As seen in Table 5, the total trip coverage done by the frequency coverages are significantly fewer in the Deutsche Bahn GTFS dataset when compare to the Swiss railway GTFS dataset. The reason for this is that the departure times are not consistent in the Deutsche Bahn GTFS dataset. The gap values/differences between departure times are not equal most of the time, and this causes the algorithm to generate less or no frequency covers between the given nodes for the Deutsch Bahn GTFS dataset. This can be improved by introducing some tolerance to the frequency finding algorithm, and this will be discussed in the future work section.



## 5.5 Evaluation between the automatically generated frequency graph and the manually created Switzerland Timetable-2017 graph

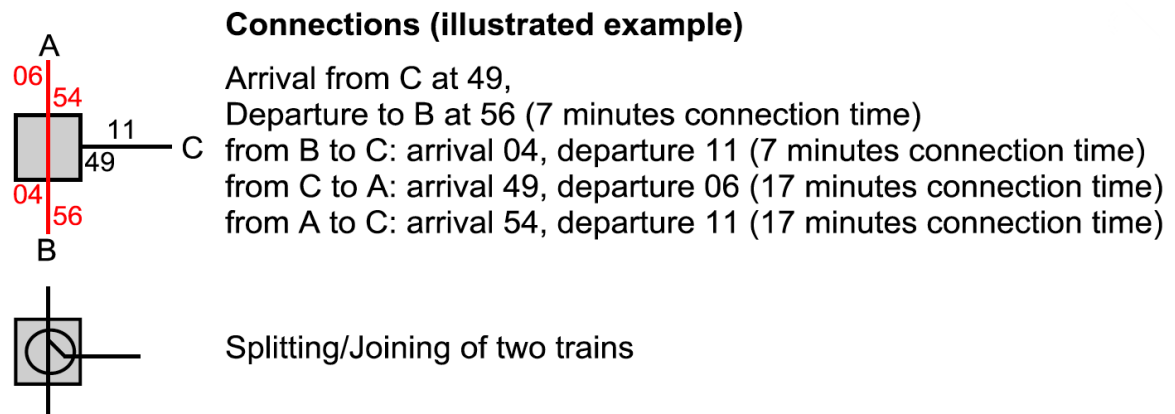


**Figure 51:** Diagram of Switzerland Timetable-2017 Frequency Graph [4]

When comparing an automatically generated frequency graph with the Switzerland Timetable-2017 graph, it is clear that the manually created Switzerland Timetable-2017 graph is superior to the automatically generated graph. Our main goal is to automatically generate a frequency graph, with little to zero human intervention that can be used as a blueprint when drawing frequency maps like the Switzerland Timetable-2017. Once all the frequency covers between the interested nodes are found and are drawn using Graphviz, then the cartographers life become easier when drawing the print ready version of the frequency map. This is because the cartographers can use the automatically generated frequency map as a reference as well as a validator to validate the manually drawn frequency map. Therefore it is clear that, the primary goal of our effort is already achieved.

The advantages of manually created frequency map over the automatically generated frequency maps are as the follows

1. The Switzerland Timetable-2017 frequency map has fewer edge crossings, this is achieved by aligning the nodes in a way such that the straight lines can cross multiple nodes. Fewer number of edge crossings makes easy to read and extract data from the map.
2. In the Switzerland Timetable-2017 frequency map, both in-connections as well as out-connections of the nodes are routed in consistent manner. Therefore, the users of the map can see how each node/station is covered by trips. Whereas in the automatically generated frequency map, in-connections and out-connections of a node are not connected. This could be implemented in the future, and more about this will be discussed in the future work chapter.
3. In the Switzerland Timetable-2017 frequency map, frequency covers such as 15 minutes, 30 minutes, are shown by drawing the frequency lines too close to each other(bundling frequency lines). This feature is not available with graphviz. Therefore with the current approach, frequencies like these are represented by drawing multiple individual lines in a way that covers one-hour frequency. Ex: Frequency covers between Zurich HBF and Brugg AG consists of 30 minutes frequency cover. In the automatically generated frequency graph, this is represented by drawing two frequency lines between Zurich HBF and Brugg AG nodes.
4. The Switzerland Timetable-2017 frequency map capable of displaying the connections and splitting/ join of two trains. This feature is not yet implemented in automatically generated frequency graph.



**Figure 52:** Illustration of connections as well as splitting/ joining of two trains [4]

As discussed, the manually created Switzerland Timetable-2017 graph has several features, which are not currently available in the automatically generated frequency graph. However the automatically generated frequency map, surprisingly close enough to the Switzerland Timetable-2017 graph, and it can be improved by investing more time and other resources into it. Nevertheless, the primary goal of automatically generating of frequency graph which can be used as a reference when manually drawing the frequency graphs is achieved here.

## 6 Future Work

This chapter is dedicated to the implementations and improvements that can be made in the future in order to enhance the quality of the automatically generated frequency graph. There were some corner cases which we already identified during the implementation and evaluation chapters. Most of these challenges can be overcome, by investing more time and resources to implement solutions which can address and resolve these challenges.

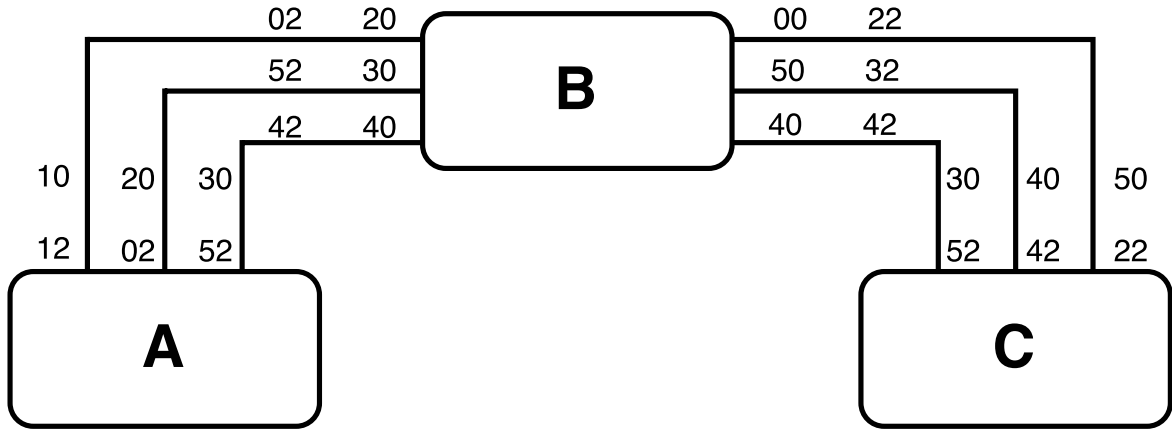
The first major improvement that can be done is by making the frequency finding algorithm tolerance in a way that it can find the frequency covers even if the departure time series consists of slightly different gap values. To understand this concept let us consider the example with the Deutsche Bahn GTFS dataset which is already mentioned in evaluation chapter.

Karlsruhe HBF the departure times from Freiburg HBF as follows.

8:57, 9:56, 10:57, 11:56, 12:57, 13:57 ..

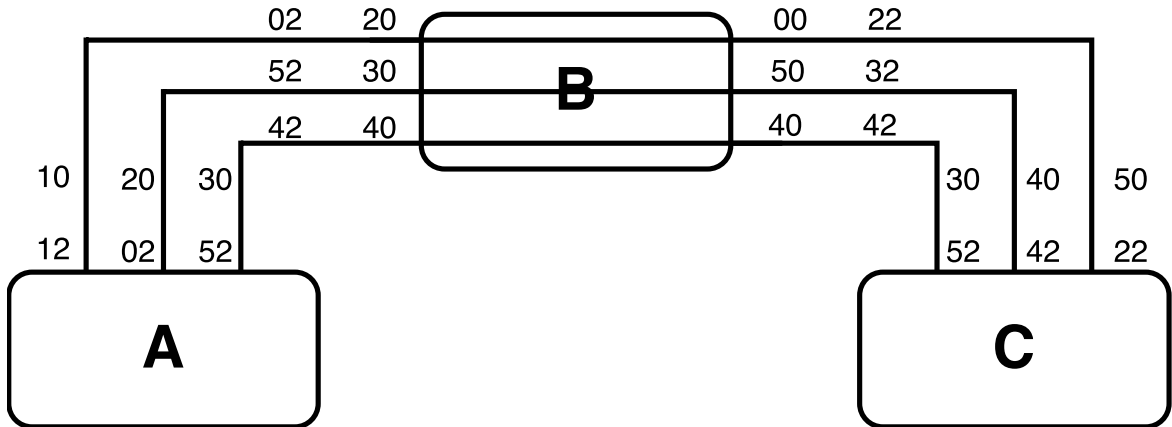
The highest difference of the gap values between these departure times is 1 minute. The key idea here is to make the algorithm tolerance enough in a way that it can ignore these minor changes among the gap values between departure times and result in this departure time series as a frequency cover. This approach will cause the algorithm to identify a large number of frequency covers in the Deutsche Bahn GTFS dataset as well as in other GTFS feeds which do not have a high consistency like the Swiss railway GTFS dataset.

The next improvement that can be done to increase the quality of the generated frequency graph is the implementation of a mechanism that connects in connections of a node to its out connection. The Switzerland Timetable-2017 graph has this feature and it facilitates the commuters by providing the information about coverage of nodes by a trip as well as its arriving and departing times to each node. To understand this let's assume there are three nodes which are node A, B and C. There are three frequency covers which cover all these three nodes. According to the current implementation, the automatically generated frequency graph will display these frequency covers similar to the following diagram.



**Figure 53:** Diagram of node A,B, C and their frequency covers

When considering the in connections and out connections of node B, it is difficult to identify the continuity of the trips that comes and leaves from node B. Therefore the idea is to connect the in connections and the out connections of a node so that it is easy to extract the information of the continuity of the trips. The following is a sample diagram that connects in connections of a node to its out connections.

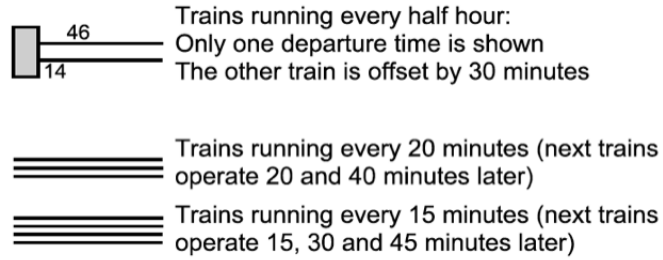


**Figure 54:** The diagram with node A, B, C and their frequency covers. In connections of node B, connected with its out connections

In connections of node B connected with its out connections. It is more clear which trips come to the node B and at what time they leave from node B. To implement something similar to the above diagram using the Graphviz tool, probably needs many modifications to the code base of the Graphviz. The reason for that is Graphviz is not designed in a way to render such a graph like this, where the edges are routed through the nodes. Its main purpose is to draw network diagrams, entity relationships, flow charts and simple graphs with different layouts.

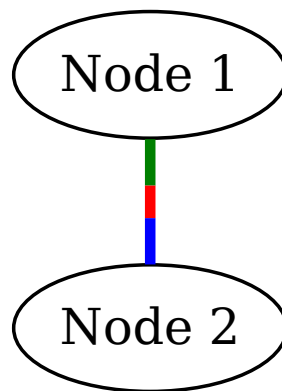
All the currently available layout engines do not support edges which go through the nodes. However, as per our observation was done on the code base, routing edges through nodes and make the connection between in-connections and out-connections might be possible to achieve by modifying the code base.

The Switzerland Timetable-2017 graph have a special way of representing frequency covers with 15 minutes, 20 minute and 30 minute gap values as follows.



**Figure 55:** Format of displaying 15 minute, 20 minute, 30 minute frequencies in Switzerland Timetable-2017 graph [4]

The approach used by the Switzerland Timetable-2017 graph is to bundle multiples edges together to represent 30, 20, 15 minutes frequency coverages. The ability to bundle of the edges between nodes is not available in the Graphviz. Therefore it is not possible to adapt the same style to represent these frequency coverages in the automatically generated frequency graph. One possible way of representing these frequencies is to introduce new edge styles using the Graphviz tool. By using different penwidth values, it is possible to draw lines with different thicknesses in Graphviz. Another possibility is to introduce multi colour edges for the special frequencies like this, so that users of the map can extract these frequencies, more easily.



**Figure 56:** Edges with different line thickness and colour combinations

Another identified problem to be solved in future is, when drawing the nodes which are geographically close to each other, the node boundaries can collide with each other. Once the node boundaries collide, that causes the Graphviz tool to render all the orthogonal edges as straight line segments, and this drastically reduces the quality of the automatically generated frequency graph. This can be prevented by making a mechanism that checks for the collisions of the nodes before drawing them and if nodes are colliding, slightly move one node further away from the other by changing their coordinates and check again for the collisions in an iterative manner. This approach will cause to change the exact location of the nodes in the frequency graph. However, this is not a problem as long as nodes are not dislocated majorly in the map. On the other hand, the important information that should pass to the commuters is how the nodes are located relative to other nodes, but not the exact position of the nodes. This is the core concept that even used in Metro Map Drawing [21].

## 7 Acknowledgments

First and foremost, I would like to thank Patrick Brosi for advising and guiding me throughout the whole process of this thesis.

My gratitude also goes to Prof. Dr Hannah Bast and Prof. Dr Georg Lausen for spending their valuable time to supervise me.

Next, I would like to thank Maxwell Sivertsen and S. Edirisooriya for helping me with proofreading my thesis.

Finally, I want to thank my family and my girlfriend Ireshika for being supportive and encouraging me all the time. I should also thank all my friends for motivating me and providing useful opinions.



# Bibliography

- [1] General Transit Feed Overview.  
<https://developers.google.com/transit/gtfs>.
- [2] General Transit Feed Specification.  
<https://developers.google.com/transit/gtfs/reference>.
- [3] geOps, “Public Transportation Feed for Switzerland.”  
<http://gtfs.geops.ch>.
- [4] Switzerland Timetable-2017.  
<http://www.bahnonline.ch/bo/18755/netzgrafik-fahrplan-schweiz-2017.htm>.
- [5] Chair of Algorithms and Data Structures of University of Freiburg.  
[http://ad.informatik.uni-freiburg.de/front-page-en?set\\_language=en](http://ad.informatik.uni-freiburg.de/front-page-en?set_language=en).
- [6] J. C. Wong, *Use of the general transit feed specification (GTFS) in transit performance measurement*. PhD thesis, Georgia Institute of Technology, 2013.
- [7] J. Wong, “Leveraging the general transit feed specification for efficient transit analysis,” *Transportation Research Record: Journal of the Transportation Research Board*, no. 2338, pp. 11–19, 2013.
- [8] H. Bast and S. Storandt, “Frequency-Based Search for Public Transit,” 2014.  
[http://ad-publications.informatik.uni-freiburg.de/SIGSPATIAL\\_Frequency\\_based\\_Search\\_BBS\\_2014.pdf](http://ad-publications.informatik.uni-freiburg.de/SIGSPATIAL_Frequency_based_Search_BBS_2014.pdf).
- [9] Google Feed Validator Repository.  
<https://github.com/google/transitfeed/wiki/FeedValidator>.
- [10] H. Butler, M. Daly, A. Doyle, S. Gillies, T. Schaub, and C. Schmidt, “The geojson format specification,” *Rapport technique*, vol. 67, 2008.

- [11] GeoJSON Specification (RFC 7946).  
<http://geojson.org>.
- [12] G. QUANTUM, “Development team, 2012,” *Quantum GIS Geographic Information*, 2014.
- [13] E. Koutsofios, S. North, *et al.*, “Drawing graphs with dot,” tech. rep., Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, 1991.
- [14] DOT Language Wiki.  
[https://en.wikipedia.org/wiki/DOT\\_\(graph\\_description\\_language\)](https://en.wikipedia.org/wiki/DOT_(graph_description_language)).
- [15] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, “Graphviz—open source graph drawing tools,” in *International Symposium on Graph Drawing*, pp. 483–484, Springer, 2001.
- [16] Graphviz web-page.  
<https://www.graphviz.org>.
- [17] Graph output formats supported by Graphviz.  
[https://graphviz.gitlab.io/\\_pages/doc/info/output.html](https://graphviz.gitlab.io/_pages/doc/info/output.html).
- [18] Command-line Invocation of Graphviz.  
[https://graphviz.gitlab.io/\\_pages/doc/info/command.html](https://graphviz.gitlab.io/_pages/doc/info/command.html).
- [19] Somayeh Sobati moghadam, “New algorithm for automatic visualization of metro map,” *IJCSI International Journal of Computer Science Issues*, Vol. 10, Issue 4, No 2, July 2013.  
<https://ijcsi.org/papers/IJCSI-10-4-2-225-229.pdf>.
- [20] S-H. Hong, D. Merrick, and H. do Nascimento, “The Metro Map Layout Problem,” *Journal of Visual Language and Computing*, 17(3), pp. 203-224, 2006, Elsevier.
- [21] M. Nöllenburg, “Automated Drawing of Metro Maps [Diplomarbeit],” *Institut für Theoretische Informatik, Karlsruhe*, 20 Aug 2005.  
<http://i11www.iti.kit.edu/extra/publications/n-admm-05da.pdf>.
- [22] O. T. D. Portal, “Switzerland GTFS data feeds for 2017.”  
<https://opentransportdata.swiss/dataset/timetable-2017-gtfs>.

- [23] Leaflet: An open-source JavaScript library for web/ mobile-friendly interactive maps.  
<http://leafletjs.com>.
- [24] Quantum Geographic Information System(QGIS) web-page.  
<https://www.qgis.org/en/site/>.
- [25] Lattice graph structure Wiki.  
[https://en.wikipedia.org/wiki/Lattice\\_graph](https://en.wikipedia.org/wiki/Lattice_graph).
- [26] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, “Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools,”  
[https://graphviz.gitlab.io/\\_pages/Documentation/EGKNW03.pdf](https://graphviz.gitlab.io/_pages/Documentation/EGKNW03.pdf).
- [27] E. R. Gansner, E. Koutsofios, S. C. North, and K-P. Vo, “A Technique for Drawing Directed Graphs,”  
[https://graphviz.gitlab.io/\\_pages/Documentation/TSE93.pdf](https://graphviz.gitlab.io/_pages/Documentation/TSE93.pdf).
- [28] Graphviz Online Repository.  
<https://gitlab.com/graphviz/graphviz>.
- [29] S. C. North, “Drawing graphs with neato,” 24 April 2004.  
[https://graphviz.org/\\_pages/pdf/neatoguide.pdf](https://graphviz.org/_pages/pdf/neatoguide.pdf).
- [30] Graphviz Edge Overlapping Issue.  
<https://gitlab.com/graphviz/graphviz/issues/1296>.

