Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Joel Lappalainen

Testing and quality assurance of a multimodal public transportation routing system

Master's Thesis
Espoo, December 31, 2019

Supervisor: Assistant Professor Mikko Kivelä
Advisor: Vesa Meskanen M.Sc.

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Aalto University
School of Science

ABSTRACT OF
MASTER'S THESIS

Author:	Joel Lappalainen	
Title:	Testing and quality assurance of a multimodal public transportation routing system	
Date:	December 31, 2019	**Pages:** 62
Major:	Software and Service Engineering	**Code:** SCI3043
Supervisor:	Assistant Professor Mikko Kivelä	
Advisor:	Vesa Meskanen M.Sc.	

During the recent decades, public transportation journey planning has become an increasingly digital process. Journey planning websites and applications are replacing the use of schedules printed on paper. Both proprietary and free license open-source public transportation routing systems have been developed. Typically these systems are used as backend services for the journey planning websites and applications.

Multimodality is an important quality of public transportation routing. Car navigators only require the capability to find routes that are accessible by car, and restrictions for when one can drive on a road are limited. On the other hand, multimodal public transportation routing systems need to take into account the available public transportation options, which often operate on schedules, in addition to the other non-transit mobility options. Algorithms used for routing in these systems have improved. As new features have been added to the systems, they have become more complex. Testing and quality assurance (QA) play a key role in the development and maintenance of these systems, but the research focused on that subject, in this context, is scarce.

In this thesis, use of known failed routing requests for improving the quality of routing is explored. Additionally, it was studied how benchmarking can be used as a quality assurance tool for public transportation routing systems. From a sample failed requests (N=10000), 90% of requests were filtered out because with varying probabilities they had failed due to known issues. Examination of individual requests showed that the filtering criteria should be improved as only one request from a sample of requests (N=30) was caused by a potentially unknown cause. The usefulness of benchmarking was examined through three use cases. One finding was that certain public transportation modes can be preferred in the Greater Helsinki and it does not significantly affect the durations of the suggested itineraries. OpenTripPlanner was used as a routing system in this thesis, but these approaches should also be applicable to other systems. More research should be done on testing and QA of public transportation routing systems as there are still open questions.

Keywords:	quality assurance, routing, public transportation
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

Aalto-yliopisto
Perustieteiden korkeakoulu

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Joel Lappalainen		
Työn nimi:	Julkisen liikenteen multimodaalisen reitityssysteemin testaus ja laadunvalvonta		
Päiväys:	31. joulukuuta 2019	**Sivumäärä:**	62
Pääaine:	Software and Service Engineering	**Koodi:**	SCI3043
Valvoja:	Apulaisprofessori Mikko Kivelä		
Ohjaaja:	Filosofian maisteri Vesa Meskanen		

Viimeisten vuosikymmenten aikana julkisen liikenteen matkojen suunnittelu on muuttunut digitaalisemmaksi. Matkaopassivustot ja -sovellukset ovat korvaamassa paperiset aikataulut. On kehitetty kaupallisia ja vapaasti käytettäviä avoimen lähdekoodin reitityssysteemeitä. Näitä systeemeitä käytetään matkaoppaissa taustapalveluina.

Multimodaalisuus on tärkeä ominaisuus julkisen liikenteen reitityksessä. Autonavigaattoreissa riittää, että autoreititys toimii, ja teiden käyttöön liittyviä aikarajoitteita on vähän. Julkisen liikenteen reitityssysteemeissä pitää taas ottaa huomioon julkisen liikenteen kulkumuodot, jotka usein noudattavat aikatauluja, sekä kävely ja muu liityntäreititys. Näissä systeemeissä käytetyt reitityslgoritmit ovat kehittyneet. Uusien ominaisuuksien myötä systeemeiden kompleksisuus on kasvanut. Testaus ja laadunvalvonta ovat tärkeässä roolissa näiden systeemeiden kehityksessä ja ylläpidossa, mutta aiheeseen liittyvää tutkimusta ei ole tehty laajasti.

Tässä tutkielmassa kokeillaan epäonnistuneiden reitityskyselyiden käyttöä reitityksen laadun parantamiseen ja vertailuanalyysin hyödyntämistä laadunvalvonnassa. Epäonnistuneiden kyselyiden otoksesta (N=10000) poistettiin 90% kyselyistä, koska vaihtelevalla todennäköisyydelle ne olivat epäonnistuneet tunnettujen syiden takia. Yksittäisiä kyselyitä pienemmästä otoksesta (N=30) tutkittiin ja vain yksi niistä epäonnistui potentiaalisesti tuntemattomasta syystä, joten kyselyiden suodatuskriteereissä on kehitettävää. Vertailuanalyysin hyötyä tutkittiin kolmen käyttötapauksen kautta. Työssä havaittiin, että pääkaupunkiseudulla voidaan suosia tiettyjä julkisen liikenteen kulkumuotoja ilman, että ehdotettujen reititystulosten kesto kasvaa huomattavasti. Työssä käytettiin OpenTripPlanner-reitityssysteemiä, mutta esitettyjä metodeja voi soveltaa muihin vastaaviin systeemeihin. Reitityksen testaukseen ja laadunvalvontaan liittyy avoimia kysymyksiä, joten tämän osalta tarvitaan lisätutkimusta.

Asiasanat:	laadunvalvonta, reititys, julkinen liikenne
Kieli:	Englanti

Acknowledgements

This thesis was originally meant to focus on optimizing routing, but after months of pondering, I realised that we lacked methods and criteria – and to some extent still do – to measure the effects of such optimization and the quality of those changes from a user's point of view. Therefore, the focus shifted towards quality assurance and testing of public transportation routing systems.

I would like to thank Mika Vuorio for suggesting me that I should do my thesis around this topic and for guidance. Also, I would like to thank Rainer Kujala for being the instructor for my bachelor's thesis, finding a suitable supervisor for this thesis and guidance early on. I am grateful that Mikko Kivelä became the supervisor and for the feedback he has given me. I would have been lost without the fruitful discussions with the thesis instructor Vesa Meskanen. Additional thanks to Teemu Peltonen, Thomas Gran, Andrew Byrd and Gard Mellemstrand for guidance, and insight on OpenTripPlanner and public transportation routing in general.

I could not have written this thesis without the opportunity I have had to work on the Digitransit journey planner project thanks to CGI Suomi Oy and the Digitransit stakeholders. I would like to thank everyone who has worked on the Digitransit or OpenTripPlanner projects over the years to provide high quality open-source journey planning software for anyone to use.

Map data copyrighted OpenStreetMap contributors and available from `https://www.openstreetmap.org`.

Espoo, December 31, 2019

Joel Lappalainen

Abbreviations and Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
GBFS	General Bikeshare Feed Specification
GeoTIFF	Georeferenced Tagged Image File Format
GTFS	General Transit Feed Specification
HSL	Helsinki Regional Transport
HSLdevcom	HSL Developer Community
JAR	Java ARchive
NeTEx	Network Timetable Exchange
NLS	National Land Survey of Finland
OSM	OpenStreetMap
OTP	OpenTripPlanner
QA	Quality assurance
SIRI	Service Interface for Real Time Information
stdout	standard output
stderr	standard error
TriMet	Tri-County Metropolitan Transportation District of Oregon
UI	User Interface

Contents

Chapter 1

Introduction

Journey planning has become less difficult in the recent decades. Before, one would have had to find information from sources that were not digitized, and the required information was often not centralized. With prior experience and existing knowledge, it is possible to drive to the destination, or take the usual bus, and hope that nothing has changed. With slightly less knowledge of the upcoming journey, paper maps and public transportation schedules can help in getting to the destination but finding the most optimal route on a longer journey in a complex public transportation network from those sources is a time-consuming and a difficult task. Because of the advances in information technology, it is now possible to plan journeys relatively fast and conveniently through use of public transportation journey planner websites and applications.

In almost any facet of life, technology has helped us to be more efficient on average. Journey planning is no exception. The performance of public transportation routing algorithms has improved in the recent years [5, 7]. But just like with any other complex digital system, journey planners, and the routing systems in the background, are at constant risk of minor and major failures. Some routing query might fail because developers have not considered a rare edge case, or all routing requests could fail due to changes in code or data. Those are two examples from the opposite sides of spectrum, in terms of impact on the userbase, but there are almost unlimited number of scenarios that can happen on a routing system, which would cause occasional or systematic failures. Multimodal routing is a complex problem, and routing results from the current solutions do not always resemble the often quite subjective view of an optimal itinerary that the users have. Additionally, from a technical point of view, the performance of these systems requires optimization. Therefore, it is essential to have testing and quality assurance (QA) tools and processes for routing systems.

Research done by Delling & al. set a foundation for how use of comparison against a "ground truth" can be used to validate quality of routing [8]. In this thesis, that approach is explored. First, the theoretical background of multimodal routing systems and testing methods is introduced. Next, OpenTripPlanner (OTP), the public transportation routing system we use, and participate in the development of, at the Digitransit journey planner project [11] in Finland, is introduced. Then, it is described how user data is utilized to improve testing and QA tools and processes, and how benchmarking of a routing software has been used in QA. The results then are analyzed and conclusions of what can be done in the future are presented. The research questions for this study are as follows: How can failed routing queries be effectively utilized

for improving routing's quality? What can be done to assure that there are no unintended changes to routing's quality during development, or in inclusion of new data? How can quality benchmarking be utilized in optimization of routing parameters' usage?

Chapter 2

Background

2.1 Routing algorithms

2.1.1 Path finding algorithms

Graphs are often used to model the data for path finding algorithms. A graph consists of nodes, sometimes referred as vertices, and edges, that create connections between nodes. Depending on the algorithm, context and terminology, different values are attached to nodes and edges. In a simple case, edges have lengths. However, cost, or weight, can also associated for traversing edges. In some algorithms, nodes and edges have labels. For example, they can be used to track if a node has been visited already. Sometimes the graphs are directed [10]. For example, if the road network is modeled, one-way roads should be taken into account.

Dijkstra's Shortest Path First algorithm, often referred only as Dijkstra's algorithm, has been a cornerstone for many path finding algorithms. It defines a relatively efficient way to find the shortest path between any pair of nodes in a graph [5]. Bidirectional search, where the search is started from the start and the end node simultaneously, can be used to find the shortest path between two nodes, while potentially less nodes are explored overall [5, 10]. Additionally, Dijkstra's algorithm can be used to find the shortest-path tree between a node and all other nodes. Unlike some other algorithms, the normal version of it always finds the shortest path, and it is relatively easy to implement.

A*—sometimes written as A-star—search algorithm can be viewed as an extension to Dijkstra's algorithm. It uses heuristics to estimate the remaining length, or cost, of the path to the goal node [5, 13]. It chooses what edges to explore based on these predictions [5, 13]. There are popular and researched heuristics, for example use of Euclidean distance for estimates, but new heuristics can be designed and implemented if need be [10]. The goal for the use of heuristics is to potentially reduce the number of nodes that need to be explored by avoiding traversing edges that are to the "wrong direction" [5]. A* should find the shortest path if the heuristics in use do not overestimate the remaining cost [13]. However, use of certain performance optimization methods, such as bidirectional search and geographical distance bounds, can put that into risk [5].

These are just two examples of path finding algorithms. There are many more, and there is research done on comparing different algorithms. A couple of good examples of such are the work by Bast, Delling & al. [5, 10].

2.1.2 Public transportation routing algorithms

There are algorithms that allow computation of a shortest path in continental sized road networks in a fraction of a second [5]. Public transportation adds complexity to route planning. Routes depend on time as they are often connected to schedules of public transportation lines [5]. Multimodality of public transportation is another problem that makes it harder to efficiently find the fastest way to get from point A to point B [5, 8].

Some algorithms are only usable in finding routes with a single unrestricted mode. These modes include car, walk and bicycle as they are not limited by schedules [5]. Although, in reality, there can be restrictions on when some area is available for walking, or some road is open for traffic, but these restrictions are sometimes ignored in routing. Then there are algorithms which can be used both, in single unrestricted travel mode routing, and in multimodal public transportation routing. Naturally, there also exist algorithms which are specific to public transportation.

Dijkstra's algorithm and A* can be used for both, public transportation routing, and for street network routing. One common approach, when using one of them in public transportation, is to use a time-dependent cost function and graph [4]. Then the edges can have a different cost based on the date and time [23]. For example, if at certain time, the next bus departs in 10 minutes, the edge has less cost than if the bus would depart in 20 minutes [23]. An alternative method is to use a time-expanded graph [24]. Then there would be as many nodes in a stop as there are departures instead of reusing one node [24].

Many of the algorithms used in multimodal public transportation journey planning require preprocessing to achieve an acceptable performance level [5]. For example, one common practice is to calculate transfer routes between public transport stops in advance. Also, if every small detail is taken into account, routing will become too complex and slow. Therefore, simplifications are needed [5].

Trade-offs exist when selecting which algorithm to use. Some of them perform better on larger transportation networks, some do not require as much preprocessing or memory, and some are simple to implement [5]. One important consideration to make when implementing an algorithm, and the data structure for it, is if it supports patching the data model. In public transportation, the schedules can change, and users of journey planners benefit from getting routing results based on updated information. Some algorithms are better suitable for handling realtime updates than others [5].

Since users of journey planners have different preferences – even the same user can prefer different qualities on different occasions – it makes sense to suggest journeys that either balance multiple criteria for each suggestion, or alternatively to offer itineraries where each itinerary is optimized by one criterion from a set of criteria. Although, it is also possible to use the balanced combination of multiple criteria as one criterion when using the latter approach.

One method is to use a single criterion—cost—which balances different preferences [8]. On the other hand, certain algorithms, and their implementations, are able to optimize multiple criteria in Pareto sets efficiently [7, 8]. These Pareto sets include itineraries that are better than all the other itineraries in at least one chosen criterion [8]. Finding Pareto sets is typically an NP-hard problem, but certain algorithms are able to find them reasonably fast in the public transportation context [7].

Algorithms in the Round-Based Public Transit Routing (RAPTOR) family are becoming more popular as they enable fast public transportation routing [7, 9]. Their high performance is based on an optimized data layout, that allows quick processing of data from memory, and in enabling parallel computing [9]. They are examples of algorithms that are specific to public transportation routing [9]. In a multimodal routing system, if you choose to use RAPTOR based algorithms, you need to use some other algorithm for ingress and egress routing on the street network. Multi-criteria RAPTOR (McRAPTOR) is a generalization of RAPTOR that allows finding itineraries that optimize multiple criteria instead of just one, and range RAPTOR (rRAPTOR) can find itineraries from a range of departure times [9].

It has been feasible to do two-criterion optimization in public transportation routing for years now, but the recent advances in public transportation routing algorithms have made it possible to do fast multi-criteria optimization with more than two criteria even on metropolitan size public transportation networks [7, 8]. Use of restricted Pareto sets instead of full Pareto sets has proven to be effective. Recently presented Bounded McRAPTOR collection of RAPTOR based algorithms are able to find itineraries in under 50 ms even with four criteria on metropolitan size networks [7]. The restricted sets still contain the same significant itineraries as the full sets while giving a boost to the performance [7]. After decades of research, the algorithms are finally reaching the level of performance and supported features, where it is becoming more difficult to improve on while still giving users of journey planners a noticeable upgrade in user experience. Although, not all implementations of these newer algorithms will be as fast as the numbers presented in research and users sometimes want more than four criteria to be optimized.

2.2 Static data

2.2.1 Transit data

A multimodal public transportation routing system requires transit data to fulfil its purpose. Since the turn of the century, there has been a push towards open public transportation data and as a result, standards have been created to provide this data [2, 18].

General Transit Feed Specification (GTFS), formerly known as Google Transit Feed Specification, is one of the transit data standards. Its development was

started by Google, Tri-County Metropolitan Transportation District of Oregon (TriMet) and a few other American public transportation agencies in 2005-2006 [18]. After that the specification has been changed through minor and major revisions [35]. The format has since spread to wide use across the globe because it is supported by many popular public transportation information systems and routers including Google Transit but its use is not mandated by laws or regulations [2, 18]. Authorities and agencies publish their transit data in this format in Finland as well.

GTFS has been intended to be general, easy to edit and easy to consume [18]. That has caused a need for extensions because the basic GTFS specification does not cover all transit data use cases. Even Google has its own extensions [40]. Another example of a proposed GTFS extension is GTFS-flex which is meant for modeling demand-responsive transportation [20].

Typically, GTFS feeds are maintained by transit agencies or authorities [2]. Some of them only maintain data for just one route, some of them maintain data for hundreds, maybe even thousands, of routes. Even though all data sets try to follow the same specifications, the data sets tend to be heterogeneous in structure. The available tools and level of expertise varies a lot between the maintainers of the data. There are also alternative ways to model the same transit network in GTFS [39]. Additionally, the maintainers might optimize their data to work on some specific system [15]. Therefore, it is not always guaranteed that the data works as intended on every system that uses GTFS data. Some parts of the data can be outright unusable, or the way something is modeled can cause suboptimal behavior. For example, one could create own GTFS route for each trip of an actual route, instead of grouping the trips under one route.

There has been some criticism towards GTFS. Mainly because of its ties to Google, focus on the American market, the use of comma-delimited text files, vagueness in terminology, lack of metadata, structural inconsistencies and the requirement for extensions to cover all use cases [15, 87]. One of the original goals of GTFS was that it would be conveniently maintainable and consumable but because of its inconsistencies, it can be difficult for some system to consume data coming from numerous different publishers [15, 18].

A new emerging standard is Network Timetable EXchange (NeTEx) which is based on a reference data model called Transmodel [29]. Each member state in the European Union, and other countries that follow EU regulations, should have had their public transportation information available in NeTEx format by December 1st, 2019 [29]. So far, NeTEx is not widely in use yet, and Finland's own Transmodel implementation is still under construction when writing this thesis [84]. Because NeTEx and Transmodel leave room for interpretation, there already exists different implementations that are not identical and leave open questions for what specific model should some agency, authority or country use [29, 84].

2.2.2 Street network

Since transit data contains only public transportation information and people do not often start or end their journeys at transit stops, routers also need information about the street network that connects the stops. A proportion of the street network data sets focus on roads accessible with cars, but for public transportation routing systems, the streets and paths where one can walk or cycle are as important, if not more important. Sometimes no public transportation information is needed to find the optimal route as origin and destination might be near each other or users of the router opt to just use non-transit traverse modes.

There are different sources for street network information. Some of those are proprietary. OpenStreetMap (OSM) is an open data initiative for providing map data under Open Data Commons Open Database License (ODbL) [58]. This license allows free usage of the data and distribution of the edited data with the same license if OSM and its contributors are credited [57, 58]. Among its data types, OSM includes street network data and transit stops.

OSM data is based on the contributions from the community. In theory, anyone can contribute to OSM. In some cases, license allows that larger existing data sets are imported into OSM [60]. This can be done either manually or through scripting [60]. Despite of the existence of OSM validation and change monitoring tools, it is possible that even larger scale changes, which contain errors, are done to OSM and taken into production use by different systems before the changes are either fixed or completely reverted [50, 62, 63].

There are three types of elements in OSM: nodes, ways and relations [64]. Ways define linear features and area boundaries by combining nodes into an ordered list [64]. However, nodes themselves can also be alone to model a pointlike features, such as bus stops. Relations are used to define how other elements construct larger structures [64]. For instance, a site type relation could be used to model a university campus area that consists of any number of elements [65]. It is possible to include other relations inside a relation [64].

Tagging is used to add meaning to elements through key-value pairs. Nodes can have tags, but it is not necessary for a node to have a tag as it might just be a point where a way changes its direction. Ways need to have at least one tag unless they are part of a relation. [64]

It is not always straight-forward to interpret OSM guidelines regarding usage of tags, which occasionally causes issues. Editors of the data expect an object with tags to be interpreted in a certain way and a system that consumes the data handles it differently. One way to circumvent this issue is to give specific instructions for OSM editors on how to modify this data so that it gets interpreted correctly by a system. Some projects that use OSM data, such as the Finnish Digitransit project, have their own pages on OSM wiki partly for this purpose [59, 61]. Obviously, there should not be large conflicts between how different systems interpret the data or else the data becomes too specific for a project or a set of projects.

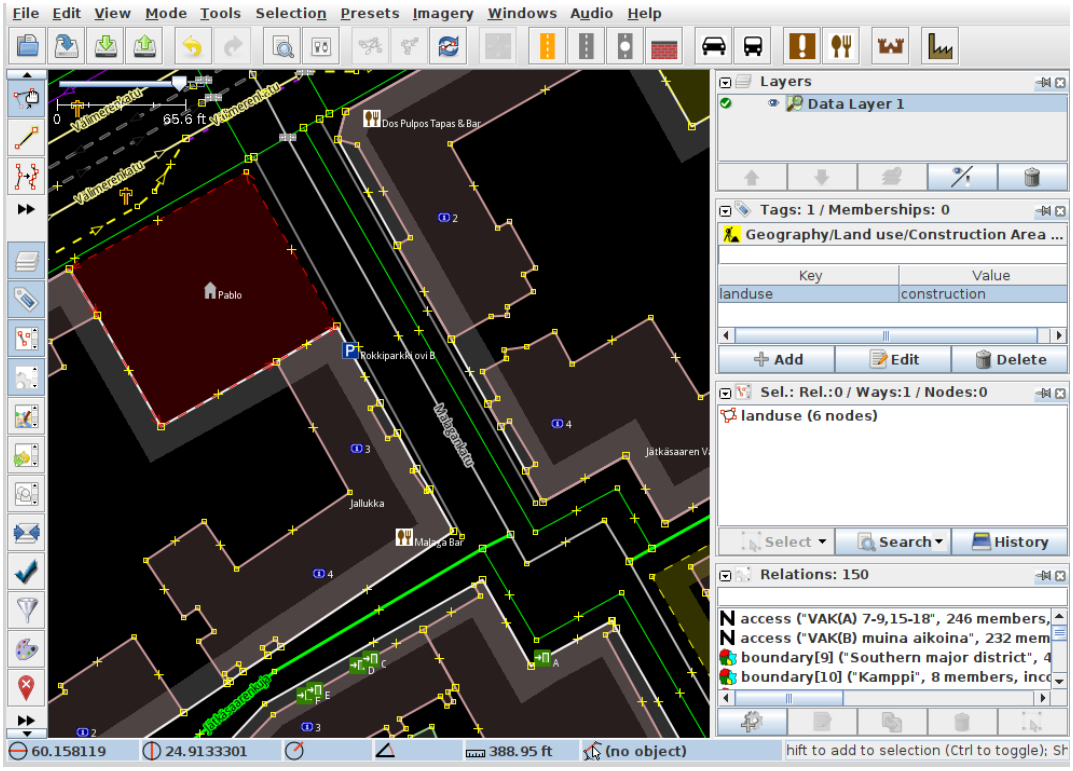

Figure 2.1: OSM data visualized on Java OpenStreetMap Editor (JOSM). OSM data is © OpenStreetMap contributors.

One of the advantages of using OSM data is that it includes a lot of data in a relatively standardized format. However, the data's spatial coverage and quality depend on the activity and expertise of the community. Studies have shown that the spatial coverage and quality of Germany's and Ireland's OSM data was almost equivalent to proprietary data sets [6, 22]. OSM data included information that was missing from those proprietary data sets and vice versa [6, 22].

2.2.3 Elevation data

People walk and cycle at different speeds when they are on level ground, uphill or downhill. Therefore, to get more realistic weight estimates for walking and cycling, elevation data can be included in constructing street network in a routing graph. Although, elevation data is not as necessary as transit data or street network data for a public transportation router. Streets or paths, in general, have been designed to not include elevation changes that would make it impossible, or nearly impossible, for people to walk on them.

Georeferenced Tagged Image File Format (GeoTIFF) is one of the standards that can be used for storing elevation data. Elevation data is stored in GeoTIFF files as pixels [82]. The distance between these pixels affects the accuracy of the data. For example, if the resolution is 5x5m, each pixel defines elevation at

the center of an area which area is $25m^2$. Naturally, if the elevation profile can be collected with better accuracy and stored in a GeoTIFF file with smaller resolution, the data will be more accurate, but the file will be larger.

National Land Survey of Finland (NLS) is a government agency that provides a digital elevation model that covers Finland in two resolutions: 2x2 and 10x10 meters [52, 53]. Both data sets are available in GeoTIFF format licensed under Creative Commons Attribution 4.0 International License [52–54]. The 2x2 meter resolution data has been collected through laser scanning the surface of the earth, but it does not cover whole Finland [53]. On the other hand, 10x10 meter resolution data is constructed from the same data as 2x2 meter resolution when feasible, but the missing areas are derived from other sources [52]. The accuracy of the 10x10 meter resolution data is worse than the accuracy of the 2x2 meter resolution for two reasons: less accurate source data on average and larger resolution. The 10x10 meter resolution data is the one used at the Digitransit project as the smaller resolution data is too resource intensive.

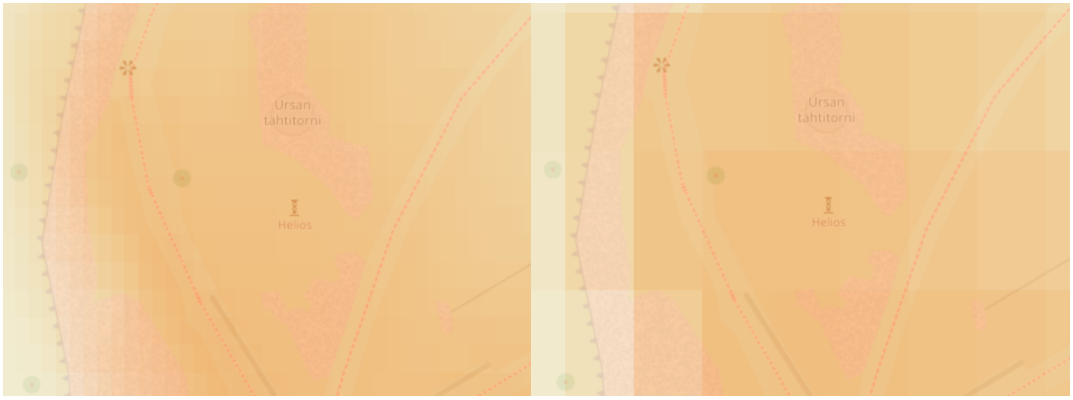

Figure 2.2: Visualizations of the 2x2 and 10x10 meter resolution NLS elevation data sets around a small hill in Helsinki on top of an OSM map layer. Areas colored with darker shades of orange have higher elevation values. Map layer is © OpenStreetMap contributors.

2.3 Realtime data

2.3.1 Trip updates

Public transportation is mostly based on predefined schedules that are often a relatively accurate representation of the real world. Bus drivers try to follow the schedules for their routes. However, there might be more people boarding a vehicle at a certain stop than expected, or more traffic on the road which causes a bus to arrive late to the next stop. Occasionally, a bus breaks down and the whole trip is cancelled, or the bus will not stop at all scheduled stops. Because

of these factors, and others that affect how well public transportation really follow the predefined schedules, realtime data formats have been introduced to support the static schedule data formats.

Since regular GTFS is meant for serving a data set that models an agency's or, a set of agencies', public transportation network, there is a lot of static information that does not need to be constantly updated. Therefore, GTFS Realtime was added as an extension to the GTFS format to allow continuous updates to systems that use GTFS data [34]. It was originally created by Google and its partner public transportation agencies, in cooperation with public transportation focused developers [34]. Since the first version, which was released in 2011, GTFS Realtime specification has been updated several times [41]. GTFS Realtime data can contain updates that affect either something that was defined in the static GTFS data, or something new.

Trip Update is one of the components in GTFS Realtime. Arrival and departure times at stops can be updated with Trip Updates, or new trips can be added that do not exist in static schedules [37]. It is also possible to cancel a trip, skip certain stops during a trip, and reroute a trip to visit different stops than defined in the static schedule [37].

There are also two other components in GTFS Realtime, Service Alerts and Vehicle Positions, but their original purpose is not to affect routing results [36, 38]. Service Alerts contain information about disruptions in public transportation network and Vehicle Positions contain information about vehicles, and their positions and movement [36, 38]. In theory, Vehicle Positions' Occupancy status information could be used to prevent a trip, if a vehicle's status is "Not accepting passengers".

As with static transit data, European Union is pushing a relatively new standard, called Service Interface for Real Time Information (SIRI), for serving realtime updates [29]. There are already realtime data feeds in Finland that use SIRI but their structures are not identical as the specification can be interpreted, and used, in different ways [49, 86, 87].

2.3.2 Bicycle and scooter rental stations

There has been a growing trend of bicycle and scooter rental networks starting their operations at cities around the world. Some of these services do not have stations where users pick up and leave their vehicles. In those cases, the vehicles can be left within a larger area where the next user can pick it up. However, there are also services which have stations, where the vehicles should locate when they are not in use.

Similarly, as for other types of open data, there is a data standard which could be used for many services. General Bikeshare Feed Specification's (GBFS) development was started in 2015 and led by North American Bikeshare Association [55]. It supports both bicycle rental networks with and without stations [55]. It is in use in over 200 different bicycle rentals systems around the world

[56]. However, many operators in this field choose to develop their own format for publishing data about their services.

2.3.3 Bicycle and car parks

There are limitations to what public transportation vehicles, and when, one can board with a bicycle. In GTFS, those limitations can be defined for trips. For instance, in Greater Helsinki, you are only permitted to take a normal unfoldable bicycle with you to commuter trains and metro, if it is not too crowded [45]. Therefore, in other cases, passengers need to park their bicycles somewhere near a bus station, for example, before boarding a vehicle.

It is common to travel a part of a journey on a car and then on public transportation, or vice versa. It can save time and lessen the issues of navigating through a city or finding a parking spot. Unless you travel on a long-distance train, or on a ferry that can hold cars, it is not possible to combine car legs, where you are not getting picked up or dropped off by someone, to transit legs without parking or picking up your car from a parking spot.

It is in the interest of public transportation agencies and authorities to promote bicycle and car park locations near public transportation stops and stations because it can guide more people to travel on public transportation. It is possible to have the locations of bicycle and car parks in a static format. Alternatively, available spaces and the status of the service can be given through a realtime feed. For Greater Helsinki, Helsinki Regional Transport (HSL) provides a park and ride application programming interface (API) with status and availability information for developers to use [46].

2.3.4 Street network updates

The traffic on roads is not constant. For instance, a popular event can affect the speed at which a vehicle can traverse certain roads. It is also possible that a road segment is temporarily closed due to an accident or roadworks. Therefore, this information can be, and have been, used in affecting which routes are given by a routing system at realtime [16].

2.4 Testing and quality assurance

2.4.1 Approval testing

There are instances, when it is not possible to create output validation logic that can be generalized to be correct on all input and output pairs. People need to validate these outputs on a case-by-case basis.

Approval testing is a testing methodology – known also by other names – which can be used when it is not possible to let a computer choose through predefined assertions if tests have passed or not [3, 80]. During approval testing,

a set of input is inserted into a software and stored with the set of output that comes as a result. Then the input-output pairs are manually examined. If the output values given the input are within the limits thought to be acceptable by a person reviewing the results, changes can be accepted [3, 80]. This method is often used for tracking behavior of legacy software [3].

This approach can also be applied to testing of multimodal public transportation routing systems as what is a good quality itinerary cannot specified accurately. However, if returned itineraries for the given input parameters can examined, it is possible to determine through knowledge of the transportation network if there are potentially better itineraries available considering the preconditions.

2.4.2 Test automation

Sometimes testing is a fully manual process. Person tests software by inserting input and analyzes the output without using any automation. It can be a good testing strategy for projects, where adding test automation would not be feasible because of limited resources or the testing does not need to be repeated. However, usually test automation can replace this manual work either partly, or completely.

The goal of test automation is to reduce manual work, increase test efficiency and consistency [12]. Manual testing is time-consuming and often expensive [12]. Unlike the time spent writing computer run tests for a software component, the time spent in doing manual tests does not necessarily lessen the time needed to test the software in the future as the manual testing of the same component can be as time-consuming the next time testing is required. Because humans are prone to errors, there is no guarantee that the tester does all the required steps to test each component on every test [12].

Test automation consists of many layers. For example, a unit test can automatically insert input and validate output of a software component. Then a test runner can run this unit test together with other tests. Continuous integration tools can be used to automatically start up a test environment where the test runner runs the unit test.

2.4.3 Quality assurance and control

Quality can be examined as a property of software, or for example, as a property of the given input and returned output of the software. Sometimes it is difficult to define what characteristics define the quality. Quality assurance (QA), and quality control, are processes used to ensure that the quality of an item stays on an acceptable level [1]. QA and quality control have slightly different meanings, but as a simplification in this thesis, QA is used as an umbrella term that covers both.

QA of public transportation routing is sometimes touched on in research that is focused on documenting the performance of existing routing algorithms,

or on introducing new algorithms. For instance, Delling & al. have presented methods and criteria they use to rate routing algorithms and routing results in their RAPTOR related papers [7, 8]. They argued that arrival time, costs and "convenience" are the most the important qualities for users, but they also acknowledged the difficulty in defining which itineraries are the "best" [8]. Itineraries have too many qualities that people prefer differently. Also, an itinerary needs to be realistic. Nobody wants an itinerary suggestion that cannot be completed in the real world.

If a model for rating quality of itineraries exists, it can also be used as part of routing code in additional to its use in external QA processes. For example, the RAPTOR based algorithms typically find a large set of itineraries during a search [7, 8, 19]. However, the users of journey planners do not want to get flooded by information. Therefore, a quality model can be used to rank itineraries as a post-processing step to filter out extra itineraries [8, 19]. It should be examined how significantly different the itineraries are so that sending homogeneous itineraries to users can be avoided [8].

Use of benchmarking against a "ground truth" has been introduced as a potential way to showcase the performance of multimodal public transportation routing systems [8]. Itineraries from different implementations of algorithms were compared against the ground truth, which in that research was itineraries from an implementation of multimodal multicriteria RAPTOR algorithm [8]. This approach is interesting as it focuses on the quality of the routing results. Often the research in this field has focused on the speed-wise performance and resource consumption of the algorithm implementations.

Chapter 3

Public transportation routing systems

3.1 Background

3.1.1 Functionalities

With public transportation, people can move from one place to another. The available public transportation modes vary based on location and time. For example, in Helsinki, one can choose between using bus, train, metro, ferry or tram but in more rural areas of Finland, bus can be the only option available. Most of public transportation is schedule-based, and routes consist of predefined stop patterns that the departures, or trips, follow [8]. A vehicle departs from a stop when it is scheduled to depart if it is running on time. However, there are also on-demand based public transportation services that do not follow a schedule or predefined stop patterns. Additionally, bike-share and scooter-share systems that function differently from more traditional public transportation systems can be viewed as part of public transportation.

Multimodal public transportation routing systems combine transit with other travel modes [8]. As public transportation does not necessarily cover the "first mile" or the "last mile" of the journey from a traveler's location to the destination, or the travel between transfer stops, routing systems fill these holes with walk, bicycle, car or scooter legs, for instance.

Routing can be configured. The ease of configuration, and available configuration options depend on what routing system is in use. These configuration options vary from query level parameters, such as what transit modes should be used, to more global settings that, for example, set limits to how long a query can be processed for. Some of the systems have support for realtime updates which can be used for patching the schedules for a trip, or the availability of bicycles at a bike-share station, for instance.

In addition to the routing from an origin to a destination, public transport routing systems can have other functionalities. For example, via-point routing is a special case of journey planning that some systems support. Also, these systems may have support for processing simultaneously one-to-many or many-to-many location routing queries that can be used in research of public transportation networks [10, 42]. As the public transportation routing systems require a lot of data to function, sometimes they provide APIs from which it is possible to fetch information about the public transportation network in addition to itinerary suggestions.

It is possible for a public transportation authority or a company to use a routing system to steer passengers into using preferred public transportation modes, routes, departures or stops. It is done by configuring these systems or the transit data in use in a desired way. This is part of people flow management that can be used to guide passengers into using transportation options that can be efficiently operated and to reduce burden from congested routes, for example.

3.1.2 Examples of implementations

There are both proprietary closed-source and free license open-source routing systems. A few well-known examples of proprietary public transportation routing systems are the ones used in the backends of Google Maps [33], Apple Maps [27] and Bing Maps [51]. These services originally had, and still have, a heavy focus on maps. However, there are also proprietary journey planning and routing software developed and used by for-profit organizations and publicly owned transportation authorities with the focus on journey planning [28, 44].

There are only a few widely used open-source public transportation routing software but additionally there are smaller projects. OpenTripPlanner [70] and GraphHopper [42] are a couple of better known examples. Then there are, for instance, open-source routers with less features that focus on a specific use case, such as capability to be run directly on web browsers or mobile devices [48].

3.2 OpenTripPlanner

3.2.1 Introduction

OpenTripPlanner (OTP) is an open-source public transportation routing software. Its development was started in 2009 on TriMet's initiative [73]. OTP is part of Software Freedom Conservancy's portfolio of projects [73]. Development of OTP is coordinated by OTP Project Leadership Committee, which has members representing organizations from the United States and Europe [73, 75]. Git is used as the version control system for development and the routing code is written in Java [71].

As a disclaimer, this thesis is focused on the HSL Developer Community (HSLdevcom) fork of OTP, which is the version used at the Digitransit project in Finland. The basic structure and algorithms of the project are largely the same as in the upstream version. However, new features have been added, and especially in regard to error tracking and APIs there are noticeable differences. These features are described in Sections 3.2.7 and 3.2.8. Additionally, a filtering of itineraries as a post-processing step has been added, which is explained in Section 3.2.2. OTP is under constant development and therefore, everything stated here about OTP's features may not be accurate anymore in the future.

3.2.2 Algorithms and data model

OTP uses an A* algorithm implementation for routing [16, 19, 76]. What separates A* from Dijkstra's algorithm, is the use of heuristics, and naturally OTP's routing uses them to optimize performance [61, 76]. There is a version 2 of OTP under development, which uses RAPTOR based algorithms instead of A* for transit routing, but that version is not studied in this thesis [19].

Since A* requires a graph to search paths from, OTP's street network and transit data is contained in a graph [61, 76]. Everything is therefore modeled either as an edge, or a vertex, or as an combination of those two. OTP's graph is time-dependent and directional [76].

Part of graph modeling is quite intuitive. For example, stops are modeled as vertices and streets modeled as edges. However, it gets more complicated when you dig into details on how transit is modeled and connected to the street network. For instance, to model a passenger boarding a vehicle, both an onboarding vertex and an onboarding edge are needed. Some of the edges and vertices are permanently stored in the graph. Others are created for temporal use at runtime and some of them can only be used within the scope of one itinerary.

There is a cost, sometimes referred as weight, attached to traversing an edge. These costs are then used to decide which paths should be searched further. In public transportation routing, it would be possible to optimize multiple criteria. For example, number of transfers and fare costs could be minimized, but OTP uses a single criterion—cost—for deciding what are the best paths in routing as with the current implementation of the algorithm using multiple criteria would slow down the routing too much [19, 61, 76]. The basic unit of cost is seconds. However, users of public transportation have preferences on how time should be spent, and what is convenient. For example, time spent walking on stairs has a higher cost than the same amount of time spent walking on a street, by default, and even higher cost if travelling on a bicycle. In addition to the cost, OTP keeps track of time spent in traveling, and it is returned to users when they fetch itinerary suggestions. Showing the accumulated cost would not be meaningful to users.

Static costs for events and cost multipliers for the spent time can be used to find results balancing multiple criteria for each itinerary [8]. However, the results from routing queries using the cost function tend to be homogeneous, and it is not necessarily what users want. Multi-criteria search with Pareto sets would allow users to pick between choices that differ more from each other and it could be communicated that in which way each itinerary is optimal. It is also extremely difficult to balance the different cost factors. If a certain factor is tuned, itinerary suggestions for some examined context can be improved while for some others they can get worse.

By default, OTP searches for itineraries which depart after a given time but it is also possible to search itineraries that should arrive before a given time [61]. To perform routing that aims towards the arrival time, OTP starts

looking for routes backwards from point B to A. This means that traversal in the graph can be reversed.

During a routing query, OTP goes through edges and vertices in a shortest path tree containing states that are linked to each other. States maintain information regarding the cumulated cost and time until that point among other things. States are edited with a state editor to create new child states, and each state can only have one child state. A new child state is created, for instance, when a street edge is traversed. State domination rules are used to determine which states can be dismissed.

OTP relies on preprocessing of data to a format that can processed more efficiently at runtime. This includes combining stop visits from trips into patterns and prerouting of transfers between stops. Preprocessing of data and the loading processes are explained further in Section 3.2.5. There are also post-processing steps in which the itineraries are constructed into a format that the users receive. Additionally, it has been observed that when multiple itineraries are fetched, some of the itineraries are objectively much worse than others. For instance, one itinerary can take one hour in total while another one includes five hours of waiting at a transit stop. Therefore, in the Digitransit project, we have created a logic that filters out these "bad" itineraries from the responses.

The search range of stops from origins, destinations and in transfers is limited as otherwise routing would be considerably slower, and more transfers would have to be prerouted and stored in memory. However, this can in some occasions mean that the optimal path, or any path, cannot be found, as it would require to travel longer distances on non-transit modes. The search range is limited by either maximum walking or cycling distance, or by a maximum pre-transit time if car is used from origin.

OTP uses different heuristics for different purposes [76]. The first of the three heuristics used by OTP is Euclidean remaining weight heuristic. It considers the distance between the current location and the destination. It is used both in street network routing and in transit routing. In street network routing, it just simply calculates the distance between points and divides it by the average velocity. When there is a possibility that a transit vehicle is boarded, it uses the velocity in public transportation and adds the boarding cost to the estimate. On the other hand, if the passenger is already on a public transportation vehicle, it does not take the boarding cost into account again.

The second heuristic is Interleaved bidirectional heuristic. It is again used for both street network and transit routing. It uses either maximum travel distance limit, or a maximum pre-transit time limit, to confine the search area depending on which unrestricted mode is being used. The heuristic is used for interleaved search from two directions – from the origin towards the destination, and vice versa. As mentioned in Section 2.1.1, there are risks in doing bidirectional search. Nodes should not be rediscovered, or else the search might not be consistent and potentially the optimal path, given the limitations in parameters, cannot be found. This heuristic ignores the time-dependency of the

graph, and all the wait times at transit stops are set to 0 as it is unknown when the passenger will arrive to the destination when the search has begun. This heuristic is used before the main graph search starts to find non-transit vertices around the two locations, and lower bound weight estimates for them. During the main graph search it additionally explores the transit vertices, and the estimates for remaining weight are based on the lower bound values gathered during the pre-search.

The third heuristic is Trivial remaining weight heuristic which always returns 0 as the remaining weight estimate, and it is not typically used during normal routing. Interleaved bidirectional heuristic is the most suitable heuristic for transit routing, as Euclidean remaining weight heuristic does not account for the fact that sometimes it is better to travel slightly away from the destination before approaching it. Also, the Euclidean heuristic prefers transit too much compared to walking as the difference in velocity is large. The Euclidean heuristic is used for non-transit searches.

After an itinerary is found, it is compacted through two-phase reverse-optimization. The goal is to find the latest possible departure trip from the origin, and the earliest possible arrival at the destination. First, a backwards search is performed to find the earliest possible arrival at the destination. Then another forward search is used to optimize the departure time. Euclidean remaining weight heuristic are used for this purpose. The process is shown in Figure 3.1.

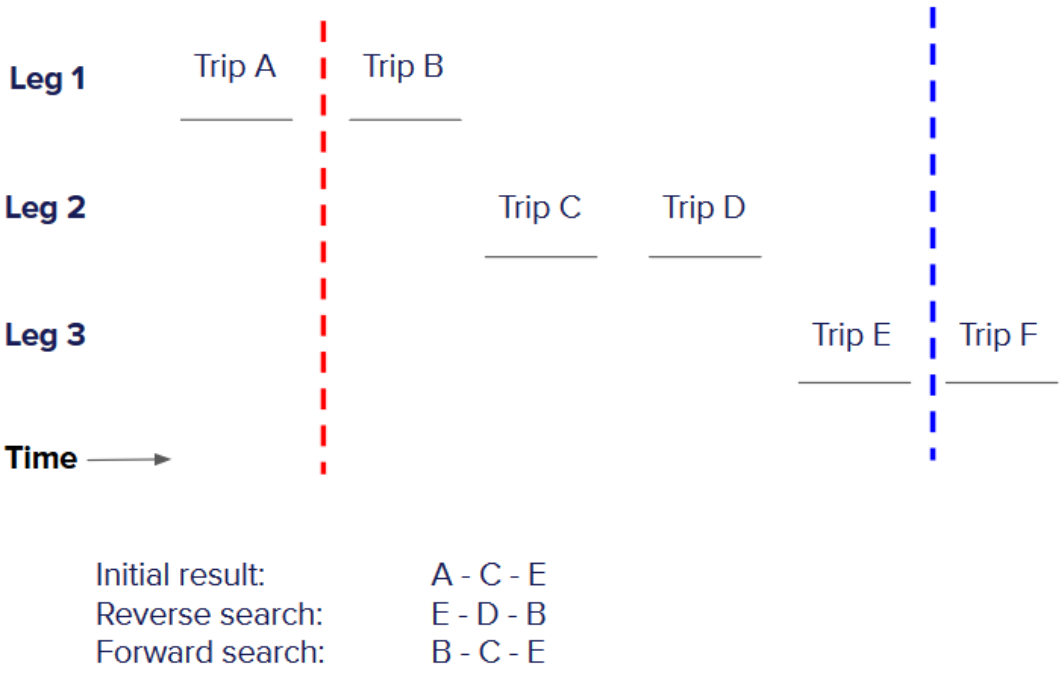

Figure 3.1: Compacting an itinerary. Copied from Mellemstrand and Gran [19]. Copyright by Entur, Norway. Used with permission from Thomas Gran.

If it is possible to use public transportation modes, OTP is often able to

find more than one itinerary. Once the first path is found, the trips used in that path are banned and cannot be used in finding more itineraries [19]. This process is repeated until OTP is able to find as many itineraries as requested, or a timeout limit is met [19]. There are downsides to this approach. First, it is not able to find multiple itineraries that purely use walk, cycle or car mode. Secondly, it is sometimes desirable to reuse the same trip for multiple itineraries that are otherwise different instead of using a completely different route. Lastly, banning a trip of a route, but not the whole route, can lead to an itinerary suggestion, where a user is told to wait extra time at a stop for the next departure of a route because the earlier departure was already used for another itinerary.

Routing with intermediate places, or via points in other words, is done by searching a route from point A to B and then another search from point B to C that starts by default at arrival time to B. These two searches are then combined into one itinerary where the two parts are connected by a leg switching edge. In the Digitransit project, we have added a possibility to query an itinerary where user spends a certain time at an intermediate location, and the time is then taken into account while traversing the leg switching edge.

The implementation of OTP's routing has become quite complex and its performance is not fully optimized [76]. Understanding and troubleshooting it is a time-consuming process [19]. Partly due to the complexity, routing contains many known and unknown bugs. Whenever a new functionality is added, that requires the core classes used in routing to be edited, there is a high probability that new bugs are introduced.

3.2.3 Supported data

OTP was originally built to support GTFS format static transit data. Support for GTFS Realtime was added later to provide realtime transit data [61]. Entur, in Norway, has been adding support for Transmodel and SIRI in their version of OTP [85]. In the future, those formats might be more widely used in OTP.

OTP uses OSM as a street network information source [61]. Only a small subset of tags is used for any purpose. For example, OTP uses the highway tag to determine how suitable a way is for bicycles [61]. On the other hand, if a way has a tag with carriage key, OTP will not use nor store this information in any way as horse carriage routing is not an available traverse mode in routing. It is possible to configure different rules for ways with different tags for all OTP instances, but also to configure specific rules, which can be enabled for an OTP instance [67]. These custom rules can be used to apply country specific speed limits for road types, for example.

Elevation data can be included in OTP [61]. This data is applied to street edges and it affects velocity on, and use of, street edges in routing [61]. It is possible to include elevation data for non-transit legs in itineraries returned from OTPs APIs.

OTP has support for bicycle-sharing networks with stations [61, 67]. Bicycle

and car parks can be added to OTP as a static import, or through an updater that keeps fetching new data from an API. There are many different updaters that can be used for setting up stations and updating their information. Most of them are for data formats that just a single operator uses but there also exists support for fetching data in GBFS format.

3.2.4 Traverse modes

Since OTP is a multimodal routing system, it supports multiple traverse modes [16]. OTP takes advantage of the extended GTFS route types to deduce transit traverse modes from data [67]. Although, it uses just a limited number of types and does not differentiate high speed rail service from normal rail service, for instance. There are multiple ways to define traverse modes used in routing. The modes can be chosen individually, or through collections of modes [67]. Transit and non-transit modes can be combined together. For example, if bicycle and train modes are chosen, it is sometimes possible to board a train with bicycle, if that is allowed in GTFS data. Additionally, the behavior of bicycle and car modes can be altered through use of qualifiers [67]. These qualifiers can enable routing that uses bike-share or car parks, for example. There are limitations to how different traverse modes can be used together. For instance, a car park leg is always before a transit leg and a bicycle leg cannot exist in same itinerary with a car leg.

OTP uses elevation data to adjust walking and cycling speed on street edges, if such data is available. For cycling, there are different optimization types available which change what factors affect the cost of traversing street edges, and in what proportions. Cycling velocity—with elevation changes taken into account—is one of the factors. Others consider the safety of the streets based on the tags used on the way objects in OSM data, and the required work in cycling on different angle slopes. For walking, OTP does not have different optimization types. OTP avoids walking, cycling and driving, by default, and the cost is twice as high as compared to equal length travel on transit. There are also other costs attached to these modes. For instance, walking on roads where cars can drive also increases the cost by default.

Wheelchair accessible routing is supported in OTP and it can be enabled in a routing query [61]. The mobility restrictions need to be considered when traversing edges. Routing on edges that have slopes steeper than the limit—4.8 degrees—in Americans with Disabilities Act (ADA) for wheelchair ramps is not allowed[81]. Also, it is possible to use the wheelchair tag in OSM to define that something is not accessible on a wheelchair and OTP respects these values, by default. Additionally, GTFS specification allows, and OTP obeys, definition of wheelchair accessibility information to stops, stations and trips [39].

3.2.5 Data loading and preprocessing

To achieve an acceptable performance, and to combine data from different formats into Java objects that can be processed in memory, OTP relies on data loading and preprocessing before requests from users can be handled. During this process, GTFS, OSM and elevation data are loaded into models and the available transfers and their walk paths between stops are searched. The result of the preprocessing is a graph. It can be serialized into a graph object file, which can be loaded into memory later when OTP server is started. Alternatively, the server is started directly after the graph build and the graph is not stored into a file between those two lifecycle phases. When the server starts, it first creates a search index for objects in the graph before accepting requests.

Since it cannot be trusted that data does not contain errors, OTP does data validation while building a new graph, which can lead to some parts of the data being dismissed or edited. For example, a trip with only one stop does not make sense and is ignored. On the other hand, if two routes have identical names, OTP adds uniqueness to the names instead of removing the routes.

It can make sense to use a wrapper project for building a new graph for OTP. This way downloading of latest data can be automated, additional data validation and modification steps can be added, and routing with real data can be tested. At the Digitransit project, we have created `https://github.com/hsldevcom/opentripplanner-data-container` for this purpose.

At the beginning of each build, it downloads the data from the previous successful build. This data is used as a seed. If download or validation fails for a new version of a data set, the seeded version is used instead. To validate a GTFS or an OSM data set, it uses OTP to build a new graph individually for each set, and if the build is successful, the data is usable. Then it edits GTFS data based on data package specific rules, if they exist. For example, the VR Group's agency identifier is edited to be "VR" instead of a number. A feed identifier, which is not part of the official GTFS standard, is added to feed information in each GTFS data set because OTP uses it to make GTFS related identifiers unique. For some data sets, GTFS stop coordinates are replaced with coordinates from OSM when feasible. Lastly, it builds a graph or graphs with a graph specific configuration file and with the new data sets supplemented with seeded data. Routing is tested before deploying the new graph and data, unless a test has failed. The implementation for the routing tests is explained in detail in Section 5.2.

3.2.6 Configuration

There are five main ways to configure OTP and some of them overlap. It depends on the context which is the optimal way to configure the system. What configuration options are available depend on if OTP is building a new graph, or if the server is running [67]. The scope, that the configuration affects, depends

on what configuration option is used. The different configuration options and how values are overridden is shown in Figure 3.2.

There are default values stored in variables in OTP's code. Some of those values cannot be changed through any other means than by editing the code. However, the default values in code will be ignored if a valid value is provided in some other configuration option. These default values exist so that every variable does not need to be configured outside of the code. If the goal is to configure OTP in a way that affects all OTP instances by default, this is the best way to do it. Some of these variables affect both graph building and a running OTP server, others affect just one lifecycle phase.

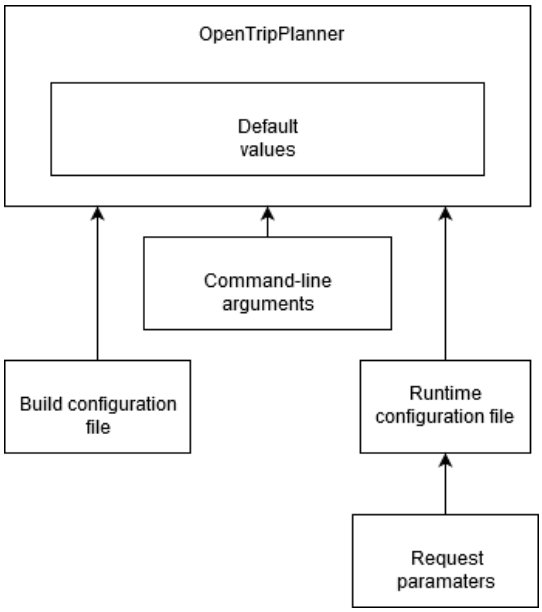

Figure 3.2: Configuration inheritance. Parameters are overridden in the direction of an arrow.

Whether OTP is started to build a new graph, or to run the server, is chosen based on command-line flags used when launching OTP [72]. For example, the flag "–build" indicates that OTP should build a graph [72]. These values cannot be overridden through other configuration means.

There are two configuration files that can be used to configure OTP at different phases of the lifecycle. First, a build configuration file affects building of a graph [67]. For example, what implementation should be used to interpret fare rules and attributes from GTFS can be defined in this file [67]. Secondly, a runtime configuration file is used to configure OTP server [67]. This file can, for instance, be used to define what realtime data updaters should be used and to set routing parameters [67]. If there is a need for OTP server instance specific configuration, or for variables that should not be defined through other means because of technical, security or usability reasons, a runtime configuration file should be used. New configuration options have been added to both

configuration files in the Digitransit project. For example, a method to prefer, or avoid, certain traverse modes over others has been added.

Finally, users of OTP's API can use a limited set of arguments in requests that override variables specified in code and in a runtime configuration file [67]. These arguments only have an effect in the scope of the request. This way of configuring OTP's behavior is especially necessary to exist in order to, for example, configure on request basis what are the origin and destination coordinates, and when is the departure or arrival time. Additionally, request parameters can be used to change routing's behavior, and the routing response, based on user's preferences instead of using the default values defined in code or in a runtime configuration file.

3.2.7 APIs

OTP has two APIs, a newer GraphQL [43] API originally developed in the Digitransit project, and a legacy, but still widely used elsewhere in the world, representational state transfer (RESTful) API [16, 66]. These APIs are in our context only used for reading data. Modifications done to the routing graph in a routing request should not persist outside of the scope of the request.

The APIs allow users to make routing requests but also to query transportation network information. For example, stops, routes and trips can be fetched from the APIs. Therefore, OTP's APIs can even be useful for systems that do not need itinerary suggestions. As OpenTripPlanner has been used for various purposes all over the world in different contexts, the RESTful API contains request paths that do not work anymore, contain vulnerabilities or allow heavy transport network analytic queries which are unwanted [66, 77].

3.2.8 Integrated testing, debugging and logging features

As a part of the OTP codebase, there are unit tests written with JUnit [14], which is a popular unit testing framework for Java projects. Surefire [26] is used as the test runner when tests are automatically run as part of the process to create an executable Java ARchive (JAR) file [68]. However, it is also possible to skip running tests while generating the JAR file [68].

In addition to the external debugging tools one can use as part of the development and testing process, there are also graphical interfaces written to OTP that can provide with additional help. First, there is a journey planning website that can be used when OTP server is running [16, 61]. It can be used to search itineraries from OTP, display a summary of them and visualize the routes on top of a map layer [61]. There are also separate map tiles, which can be fetched from OTP, that visualize street network graph's wheelchair accessibility, cycling safety factors or traverse mode permissions (walk, bike and car) [61, 79]. These layers can be shown on OTP's own trip planner user interface (UI), or be fetched to some other UI [61, 79]. Lastly, there is a visualizer component that has various features, some of which do not work

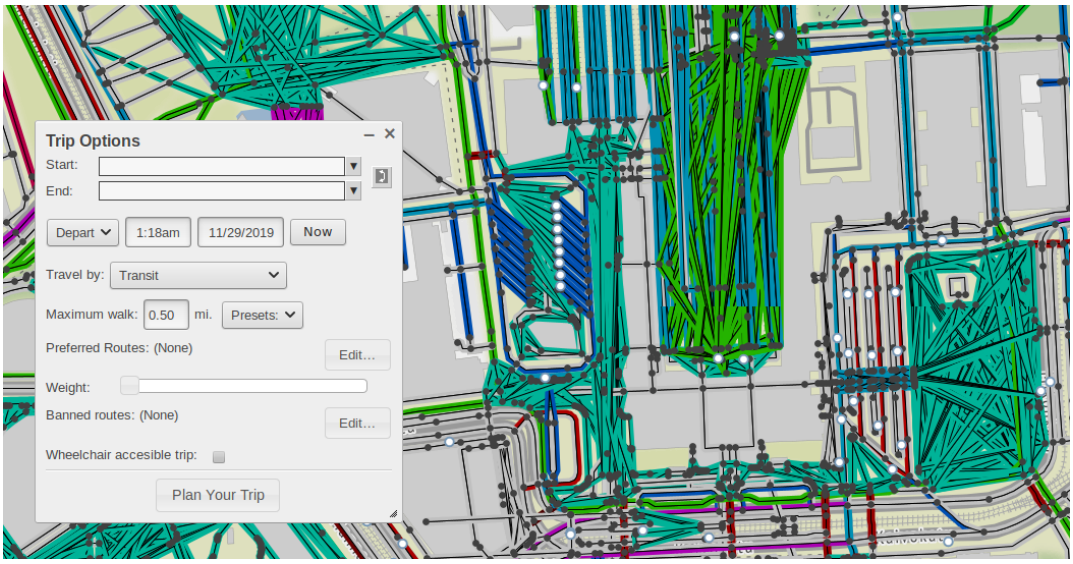

Figure 3.3: OTP's trip planner user interface (UI) with bicycle safety factor visualization shown

anymore. One of its core features is an animation of how edges are explored during routing.

OTP logs warnings and errors in standard output (stdout) and standard error (stderr). This happens both, during the graph build, and when the server is running. In addition, in the Digitransit project we have integrated Sentry, an error tracking software, to send information regarding error events to Sentry's servers where the events are categorized [30].

OTP uses annotations in graph building to mark potential issues faced during graph building [79]. A summary of these is printed at the end each build [79]. When a new graph is built for OTP with the wrapper project, the stdout and stderr logs of the data build are stored, including OTP's logs. The data build also outputs lists which GTFS stops were connected to OSM stops during the build, and those which were unconnected.

3.2.9 Performance factors

The resources which are available for the routing software, OTP in this case, affect the performance and quality of routing. More suitable hardware allows higher throughput for processing queries, and the quality of routing is directly affected by performance as it is necessary to have some sorts of timeout limits for itinerary search times in most use cases. There are three resources that affect OTP's performance during runtime: network, memory and central processing unit (CPU). Out of those three, it is known that memory and CPU are more likely to be bottlenecks than network.

There are some hard limits for the host that the routing system is running on. For instance, graph should reside in memory but it cannot be loaded there

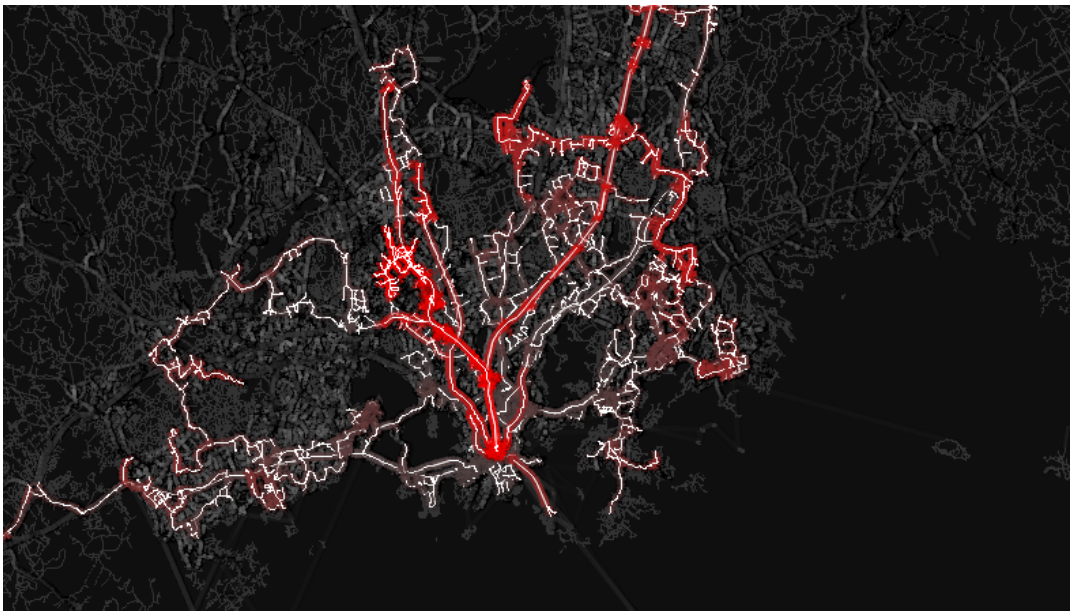

Figure 3.4: OTP's visualizer UI after a processed itinerary search

if there is not enough memory allocated for Java [72]. Similarly, there needs to be enough processing power to handle routing requests. After a critical point is reached and routing is functional, throughput can be improved by scaling up the resources horizontally or vertically. In this context, horizontal scaling means adding more CPU cores of the same quality. On the other hand, vertical scaling means that the resource is replaced by a product with better performance. In practice, this means memory with faster access times, or CPU with better single-thread performance, for example.

Naturally, the performance can also be improved by optimizing configuration or code. For example, the maximum walk distance limit can be reduced to allow faster search times. Data model can be minimized so that it fits in less space, and the memory layout can be optimized to allow better access times from memory and from CPU cache. Code can be refactored to use less operations processed with CPU and to allow processing of more operations in parallel. Throughput of routing can also be improved by duplicating routing instances and putting them behind a load balancer.

Chapter 4

Methods

4.1 Common research context

For production use of OTP, there are three sets of configurations and data—both static and realtime—used in the Digitransit project. One is meant for HSL area, Greater Helsinki in other words, use. The second one, referred as Waltti, is for 16 other cities, and their surrounding areas, in Finland. The final one contains almost all the data that the other two have in use, but also additional data from, for instance, the national railway company VR Group.

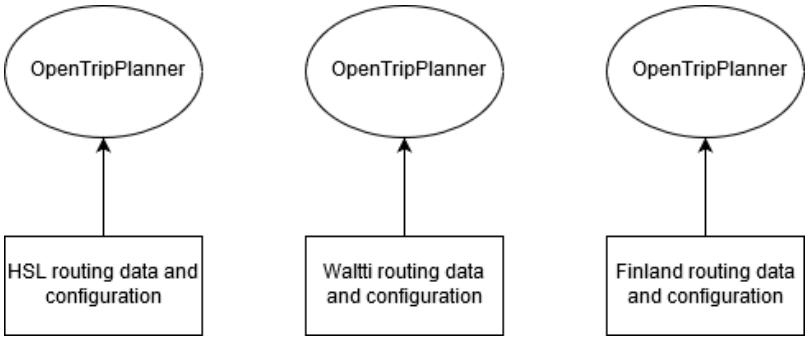

Figure 4.1: Simplified routing instance architecture

These are typically the different types of OTP instances we want to test and have data for and from in the Digitransit project. Therefore, those are in the focus in this thesis. Realtime data was not used in the test setups here as it adds a variable that cannot be easily controlled. However, when data that originated from the production instances was used, it was potentially affected by the use of realtime data.

4.2 Processing of known faulty queries

Sentry groups errors by their fingerprints [31]. Additionally, it is possible to see how many of the events for the same issue had some value for certain "tag" [32]. It can be checked, for instance, if certain issue occurs more often in OTP instances configured to use certain graph than in other instances. However, this functionality is slightly limited, and therefore, it makes sense to fetch the data from Sentry through its API and process the data elsewhere. To this end, a tool was created, and its implementation and usage were documented.

The Sentry event handling in OTP had been modified in the Digitransit project earlier so that the routing parameters, and debug data produced by OTP, are sent to Sentry when no itineraries are found for a routing query. A sample of these events was then fetched from the API with the tool created for this study. This data was used to group and visualize the parameters that were present in those queries. Additionally, a sample of the queries was resent to OTP to see if it was still unable to find itineraries. If the problems could be reproduced, the potentially causes for the issues were searched.

Before grouping of parameters from queries was started, known patterns that can cause no itineraries to be found were identified. Then queries that most likely failed due to those patterns could be programmatically filtered out. That by itself is a useful process as if individual queries are examined to pinpoint problems, browsing events with known problematic parameters is not desired.

Parameters from these faulty queries can be grouped by exact values or through fuzzy logic. For example, there are only limited number of options for the maximum number of transfers. On the other hand, for some parameters there are too many options to group them by exact values. Those parameters with only a limited number of options available, or in use, were combined into sets of combinations. The total number of combinations and the number of occurrences for each combination were calculated.

Coordinates that are relatively near each other were clustered, and those that could not been clustered were stored as outliers, so it would be potentially possible to see if there are larger areas that cannot be accessed because of errors in data or code. The coordinates and coordinate clusters were visualized. QGIS [25] was used to this end.

The overall goal was to focus on identifying and fixing issues that still existed when the events were fetched from Sentry. Some of the events were potentially not relevant anymore as the data or code had been changed since then, and the underlying problems had been fixed. Therefore, data from the date when the issues were attempted to be produced was used instead of the data from the date when a particular event was generated. This also simplified the process as it was not required to test against multiple versions of OTP.

The hypothesis was that identifying issues through visualization and grouping is slightly uncertain. However, manual examination of the queries that were left after the filtering was expected to yield results.

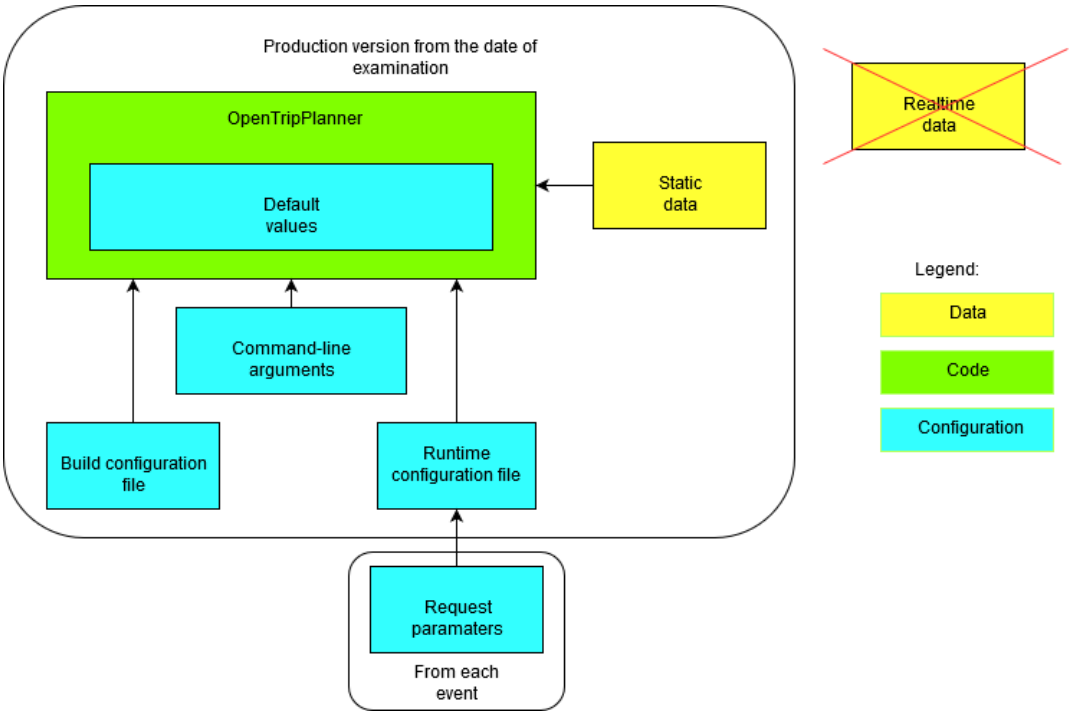

Figure 4.2: Context for studying individual events

4.3 Routing quality benchmarking

To enhance understanding of routing's quality and to track how it evolves, metrics are needed to measure it. Additionally, a way to benchmark how different versions of a routing system, or even completely different routing systems, compare against each other in terms of these metrics is required. The differences between versions can be in code, how they are configured, or in the data they use. Therefore, a software that can automatically, or semi-automatically, be used to benchmark routing based on different metrics is needed.

In this study, a quality benchmarking tool, and the changes done it, are documented. Additionally, its usage for three use cases is described through practical examples. First, to support development work. On top of other testing methods, it can provide information that adds confidence in that the changes are working as intended, or alternatively it can bring forth potential issues. Secondly, it can be used as part of a data building process to test that the quality of routing has not dropped. Lastly, to aid the process for choosing how a router should be configured. Information on how changes in configuration affect individual routing results, and results overall, should be gathered before it is possible to estimate what is the "most optimal" configuration or set of configuration profiles. The usefulness of this approach was tested by changing a small set of parameters, which affect the cost of traversing edges, and the results were visualized and analyzed.

Information was gathered on what origin and destination coordinates users

of journal planner websites have used. This data comes from Matomo, a web analytics software that was already in use in the Digitransit journey planner's UIs [17]. This information is useful as it provides insight into what should be tested if the focus is on improving user experience. The coordinates are then used in the benchmarking tool to mimic queries from users while testing the quality of routing.

For this study, data from 14 of these UIs was used to test routing of OTP instances. The chosen UIs are all websites and they use the same UI project that is merely configured slightly differently for each instance and they all use OTP for routing. Each site is used in different parts of Finland, and therefore, the coordinates from users should be mostly limited within the geographical area of the intended area for each UI. Data from one of the sites, `https://reittiopas.hsl.fi`, was used to test HSL routing instances. The data from the remaining sites was used to test Waltti and Finland OTP instances' routing. For the latter, the data from HSL area was also utilized.

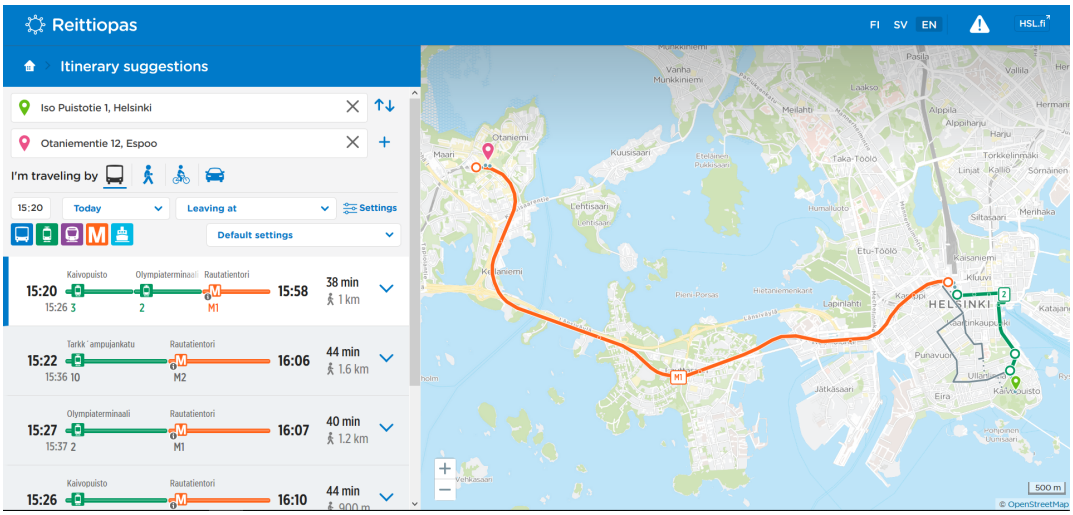

Figure 4.3: A screenshot from https://reittiopas.hsl.fi after a successful itinerary search.

Chapter 5

Results

5.1 Faulty queries filtered, visualized and grouped

The tool that was created for fetching and processing Sentry events is `https://github.com/hsldevcom/digitransit-sentry-analytics`. It processes the data and generates a summary of the error events. In addition, it creates a file containing coordinates from individual queries, and another one for coordinates of clusters, and outliers, created based on the coordinates from queries. It stores the number of queries that used each coordinate, or coordinates belonging to each cluster, together with the coordinates.

A small number of known reasons to why no routes are found sometimes were identified. Events that were potentially caused by these known issues were automatically filtered out from the fetched data as there is more interest for the unknown causes for errors. Queries which had invalid coordinates as either origin or destination were removed because no itineraries can ever be found for these. If a search has coordinates too close to each other—30 meters was used as the limit—it can cause them to be projected on the same edge which causes an error in routing, and therefore those entries were filtered out. Coordinates, which were clearly outside of the areas with good data coverage for each routing instance type, were removed. Also, because it is not guaranteed that the GTFS data includes historical data, queries that had the aimed arrival or departure time over one day in the past from the date when the event was triggered were removed. Fare restrictions can cause routing to fail because it is possible that no routes will be found with the selected fare, and therefore those entries were discarded. Lastly, queries which allowed usage of transit modes, but did not include bus, were left out since stops for other modes are not as widely available and it can cause routing to fail because the search range for stops from origin and destination is limited.

A sample of 10000 sentry events regarding no itineraries found in routing was fetched. These events were generated between November 5, 2019 and November 27, 2019. Of the error events, 13.6% were from the Finland routing instances, 51.1% were from the HSL instances and 35.4% were from the Waltti instances. In these events, 53 combinations of routing parameters were used. The result of the filtering process is presented in Table 5.1.

The different origin and destination coordinates used in different routing instance types were clustered. Summary of the coordinates, clusters and outliers can be found in Table 5.2. Coordinates from all routing instance types are visualized in Figure 5.1. A visualization of the clusters and outliers can be found in Figure 5.2.

Category	Occurrences	Percentage of events
From or to has invalid coordinates	3380	33.8%
From and to are close to each other	2440	24.4%
Coordinates are outside of router's area	392	3.9%
Search time too much in the past	1820	18.2%
Ticket limitation	22	0.5%
Bus not in public transportation options	1110	11.1%
At least one known potential cause	**9000**	**90.0%**
Other causes	**997**	**10.0%**

Table 5.1: General Sentry errors summary from November 27th, 2019

Category	Coordinates	Clusters	Outliers
HSL origins	386	21	46
HSL destinations	355	21	34
Waltti origins	235	24	76
Waltti destinations	233	26	104
Finland origins	97	6	70
Finland destinations	97	11	66

Table 5.2: Coordinates, clusters and outliers

A sample of events left after the filtering was randomly chosen for closer examination. However, 10 events from each routing instance type were included so it could be examined if there are major differences between events from different instance types. For each event, a routing request was created with the stored parameters. Those requests were then sent to OTP. If no itineraries were found, the potential root cause for it was searched. OTP version and data from November 27th, 2019 was used for this testing as it was the date when the data was fetched from Sentry. The results are shown in Table 5.3.

Category	Occurrences	Percentage of events
Could not be reproduced	9	30.0%
From and to are close to each other	5	16.7%
No stops within search range	15	50.0%
Potentially unknown issue	1	3.3%

Table 5.3: Sample events categorized

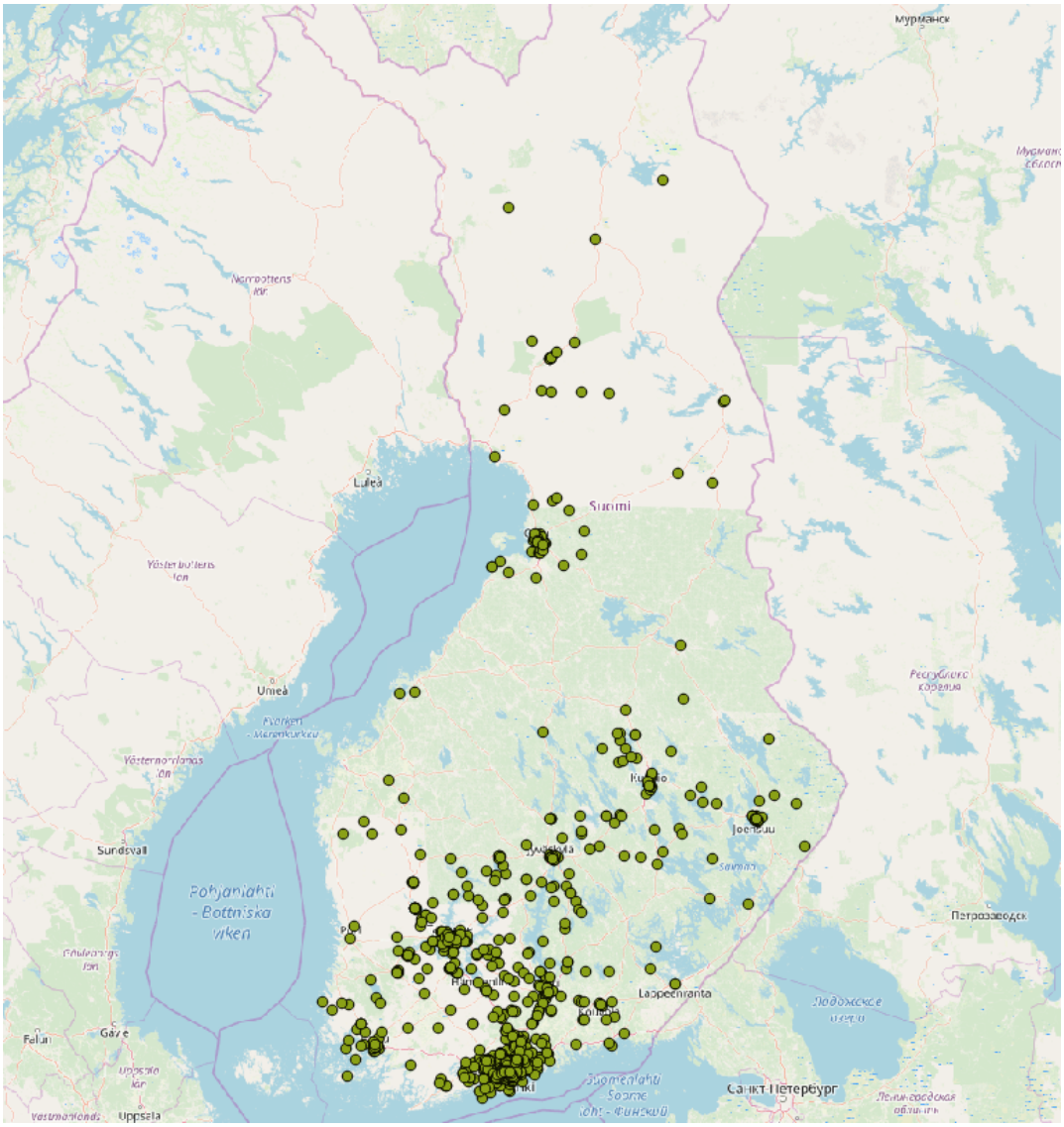

Figure 5.1: Coordinates visualized with QGIS on top of an OSM map layer. Map layer is © OpenStreetMap contributors

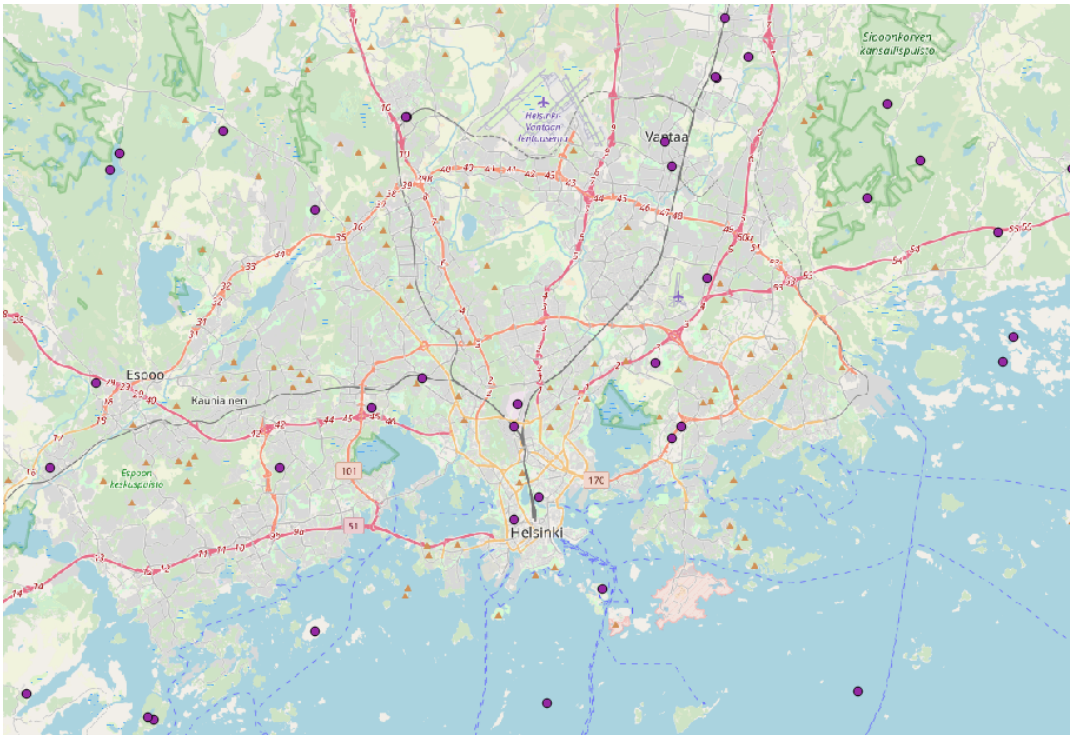

Figure 5.2: Coordinate clusters around Helsinki visualized with QGIS on top of an OSM map layer. Map layer is © OpenStreetMap contributors

5.2 Routing quality benchmarking tool

5.2.1 Implementation

We have continued development of OTPQA [74], a project written in Python and originally started by OTP developers in 2012 for the purpose of testing OTP's routing, as part of the Digitransit project in a fork under HSLdevcom organization in Github. The project is open-source and licensed under GNU General Public License v3.0 [69].

The project originally only contained a method to generate coordinates based on GTFS stop locations and highway locations from OSM data. Support for fetching origin and destination coordinates from Matomo has been developed in the Digitransit project to replace the existing way of generating coordinates. Also, a developer from the Digitransit project has integrated into this process the clustering of the fetched coordinates that reside close to each other into one coordinate. The coordinates, that are clustered into one coordinate, each have a weight based on how many times the coordinate have been used in routing, and that is used in deciding the location for the cluster coordinate. These coordinates should more accurately correspond to what the users will query, and therefore those are used for every use case of OTPQA in the Digitransit project and in this thesis.

The coordinates were then used for building requests, which were later sent to fetch itineraries from OTP. As it is known from which website each coordinate is from, the requests are grouped under websites that the coordinates originated from. These collections of requests are linked to an instance type, or to instance types, that should have data for the areas of these coordinates. This way different sets of requests can be used against different routing instances and to pinpoint the geographical area in which the routing is not working well enough.

Another feature added as part of the Digitransit project is a new way to compare two sets of routing results. It can be used for benchmarking based on various metrics. As a prerequisite, the two versions of OTP need to be profiled with another component of OTPQA that already existed with the project. Prior to this thesis and during it, this component has been modified to be more configurable, and to output more data that can be used in comparisons. The profiler was used to send routing requests to an OTP server that was located on the same host, as sending queries to another host adds an unnecessary variable that can change results. The profiler outputs a run summary file which can used for the comparison.

Metrics for benchmarking were chosen based on what information is available from OTP's output, and what is needed for measuring quality. The list of metrics that are used in comparing each itinerary search result is listed below in the order they were added and from those, the first two metrics were added prior to this thesis by another developer.

1. The first itinerary's total duration (s)
2. Was an itinerary returned (true/false)
3. Number of returned itineraries
4. Number of different traverse modes included in one response
5. Number of legs in the first itinerary
6. Average walking velocity (m/s)
7. Average cycling velocity (m/s)
8. Time it took to perform the itinerary search (ms)
9. Number of timeouts during the itinerary search
10. Number of trips within the first itinerary

The comparison outputs a summary of differences regards to the chosen metrics between the two input run summaries from profiling. It prints a line of information about the differences for each metric used in the itinerary level comparison if the difference between the two input files was above a threshold, which can be configured for each of those metrics.

The first two metrics are the only ones that are enabled by default. Others are disabled by default and can be enabled with use of command-line flags. This way the focus can be put on the information that is relevant for the QA context at hand. For instance, if only transit related code is changed in OTP, the average walking and cycling velocities are not particularly relevant metrics. Using extra metrics increases the amount of output from the comparison. Although, otherwise it does not cause any harm, and the comparison is always a quick operation compared to profiling. Different magnitudes of change can be observed by changing the thresholds. For example, if it is desired to find the edge cases where the average cycling velocity has changed considerably, a higher threshold can be used. If the focus should be put more broadly on all changes in cycling velocity, a lower threshold is used.

We have created as part of the Digitransit project a wrapper script for the profiling component that can be integrated into a test pipeline. It runs the profiler with requests generated for routing instance types given as parameters. If it considers over 10% of the queries to have failed, it exits with error code. The query can be considered as failed if no itineraries are found. This script also produces a report that lists queries for each test set, if itineraries were found for those queries, and the success rate of tests per test set. When new data is fetched from Matomo, this component is used to test that the data is still usable for testing routing and does not cause too many itinerary searches to fail. Only after that is validated, the new data can be taken into use for all benchmarking use cases.

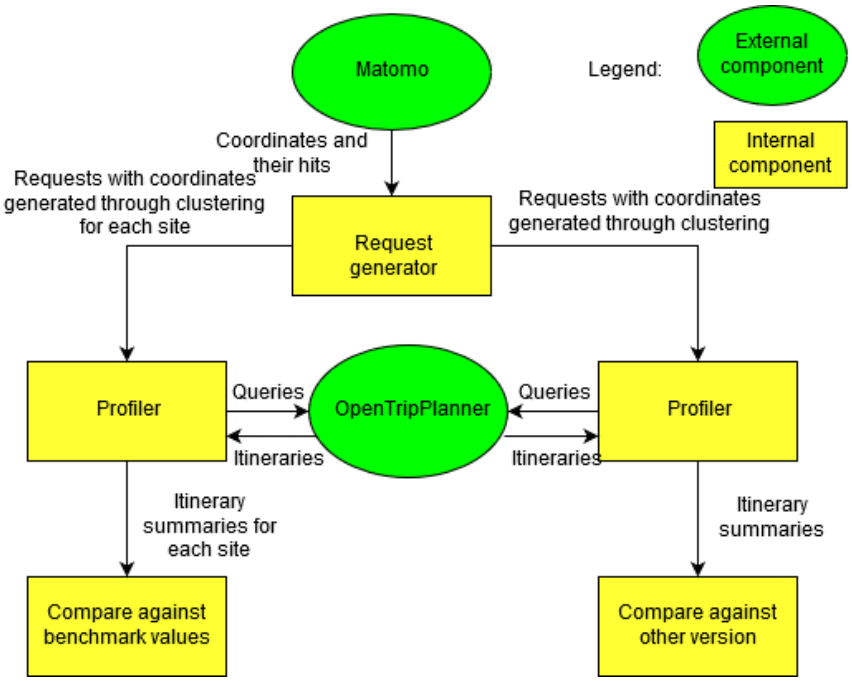

Figure 5.3: Process diagram for OTPQA's two benchmarking methods

5.2.2 Case 1: tracking quality during development

OTPQA has been used to support development of OTP in the Digitransit project, and to review changes done by external developers. One example of such usage was a review of a change to how walking velocity was calculated considering elevation changes. The algorithm was changed to use Tobler's hiking function. The function's formula is

$$W = 6e^{-3.5|\frac{dh}{dx}+0.05|} \tag{5.1}$$

where W = walking velocity and $\frac{dh}{dx}$ = slope [83]. This means that the maximum walking velocity is on downhill where the slope is -2.86 degrees. OTP then calculates edge's effective length using this velocity. The effective length can be at maximum three times the normal length to avoid issues caused by inaccurate data. The time and cost it takes to travel an edge are calculated using the effective length and the normal walking velocity.

OTPQA was used to check what the average walking velocity was after this change. It was observed that the average walking velocity went up from 1.10 m/s to 1.46 m/s when the suggested walking velocity in the request was 1.22 m/s. This result was not expected as the velocity should at maximum be 20% faster compared to the velocity on level ground, and the itineraries do not consist of walk legs that are purely on the optimal angle downhill. After this was noticed, the algorithm was shortly fixed to output more realistic walking speeds on edges.

5.2.3 Case 2: QA as part of graph build automation

OTPQA was integrated into our process for building new graphs with updated data for the use of OTP instances in our development and production environments. At the end of the build process for each graph, the graph is loaded into OTP together with a runtime configuration file, which could also have potentially changed since the previous successful graph build. A set of 200 test queries is then run per each site that is relevant for testing the graph using the clustered coordinates. The departure time for the queries is 14:00 local time one week from the moment of the test. If the date is on weekend, or on a day with special public transportation arrangements, one day is added to the date until it is a normal working day. If the test fails because less than 90% of the queries get itineraries as response, the graph build has failed, and the new graph is not deployed.

The most common problems that the test can catch relate to quality of GTFS data. Publishers of the data have either made mistakes while updating the data that reduce the available departures, or they have not yet added new schedules while old ones are expiring. The test rarely fails due to issues in OSM data, or in configuration.

5.2.4 Case 3: configuration optimization

Finding an optimal routing configuration is a difficult task. Often the demand for changing configuration arises from witnessing an itinerary suggestion that does not correlate well with what was expected. It might be possible to get the desired suggestion after changing a parameter, or a set of parameters. However, as a side effect some other itinerary suggestions are not optimal anymore. Therefore, it was tested how statistics from OTPQA could be utilized in configuration optimization.

In this case study, the value of a walk reluctance parameter was changed in a HSL routing instance. It is a multiplier which affects the cost for walk legs, and it is used for controlling how much less desirable walking is compared to time spent on public transportation. The "ground truth", or the benchmark in other words, used for this study was the code, data and configuration—can be found in Appendix A.1—from production HSL routing instances without the use of realtime data. This version of OTP was profiled and all the other versions of OTP were compared against the results from that profile run. Each version was profiled by doing 800 routing queries with walk and transit traverse modes. The value of the walk reluctance parameter was increased and decreased by 0.25 at a time. The focus was on two variables: duration of an itinerary and number of transit vehicle boardings.

First, only the walk reluctance value was changed. Then all the traverse mode preference settings, which are normally in use, were removed from the configuration and the same values for the same values for the walk reluctance parameter were tested as in the first test case. Lastly, the traverse mode pref-

erences were included again but an extra cost for boarding vehicles was set to zero. Results were visualized on a Cartesian coordinate plane. A visualization of the first and second case is shown in Figure 5.4 and a visualization of the first and the third case is in Figure 5.5.

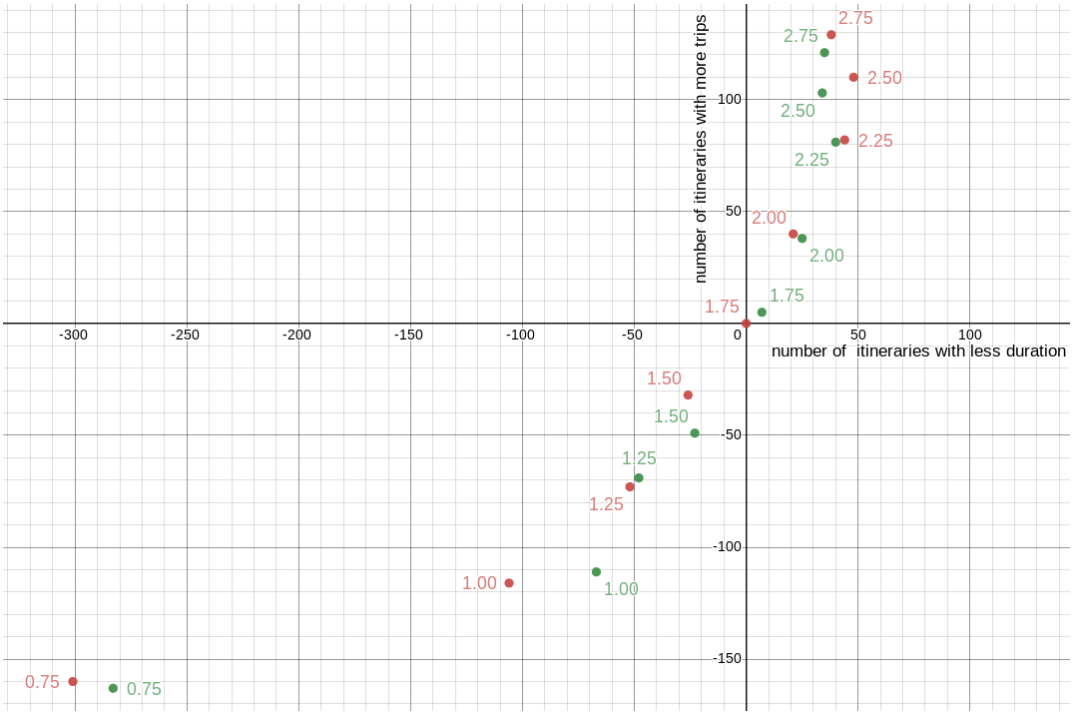

Figure 5.4: Effects of changing a walk reluctance parameter's value from the current value—1.75—shown on a Cartesian coordinate plane. Red points are from OTP instances with otherwise normal configuration and green points are from OTP instances configured to have no cost reductions and increases based on traverse mode preferences. All points are results from comparing profile run results from OTP instances with aforementioned configuration changes against profile run results from an OTP instance with no configuration changes. x is the number of itineraries with less duration with new configuration subtracted by the number of itineraries with less duration with the benchmark configuration. y is the number of itineraries with more transit vehicle boardings with new configuration subtracted by the number of itineraries with more transit vehicle boardings with the benchmark configuration.

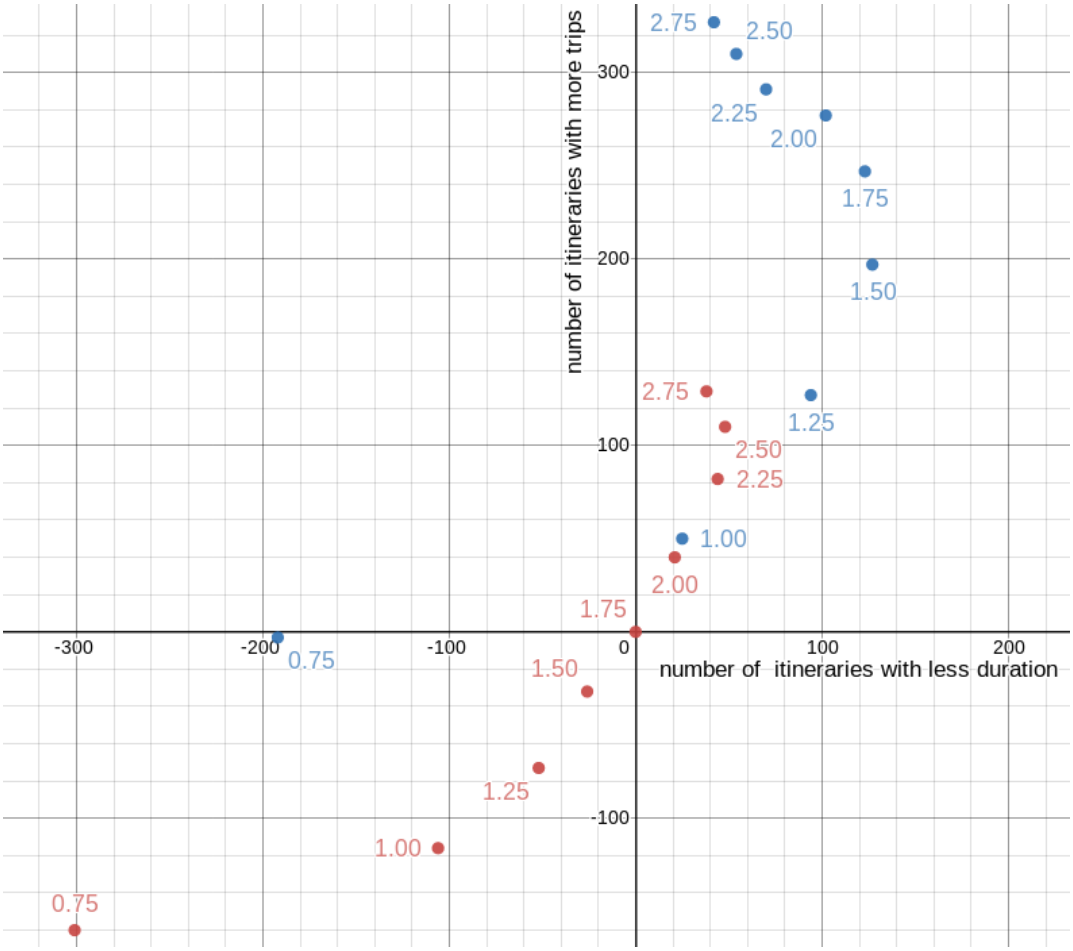

Figure 5.5: Otherwise same visualization as in Figure 5.4, but instead of green points there are blue points, which are from OTP instances configured to have zero extra cost for boarding vehicles.

Chapter 6

Discussion

6.1 Use of known errors in QA

The project that was created for processing Sentry events made it more feasible to examine individual queries that were failing, but also the grouping and filtering of results is helpful in understanding the bigger picture regarding these errors. For instance, the percentages of error events that were generated by each routing instance type differ considerably from the percentages of requests going to each type. HSL routing instances handle 74.7% of all queries but for the filtered error events, its slice of the cake was only 51.0%. Naturally, the other routing instance types were proportionally more prevalent in the error events. Finland routing instance type's share of all requests became over two times bigger from 6.7% to 13.6%. Although, there is some inaccuracy in this comparison as no data is available on what percentage of all queries are routing queries, but there are no known reasons to why the proportions of routing queries from all queries should differ considerably between routing instance types.

There are at least three important reasons why routing for these larger areas had proportionally more failing queries than the routing for HSL area. First, the coverage of public transportation in HSL area is far better than it is in less populated areas. If there are no stops with available departures near the intended origin or destination within the maximum walking or cycling distance limitation, routing will fail. Secondly, HSL routing contains OSM data for a smaller area and consumes less GTFS data sets compared to other routing instance types. Often, the maintenance of the OSM and GTFS data for less populated areas is not on the same level as it is for HSL area data. Consuming many GTFS data sets can create problems because despite using the same specification, the data is not always modeled the same way and more attention is paid to making sure that one larger data set is being consumed as intended compared to a set of smaller ones. Lastly, Finland's graph is more complex than HSL's which slows down routing and occasionally leads to timeouts. However, the timeout limits have been configured to be higher for the nationwide routing.

The used filtering criteria varies in probability to cause issues in routing. If at least one of the coordinates is null, this causes routing to always fail. On the other hand, if a user has selected only tram and walking as possible traverse modes, it can be the reason why no routes were found if at least one of the coordinates is far away from a tram stop. However, it is also possible that the coordinates are near tram stops and the reason why no routes were found is something else.

It would be possible to go through a part of the queries that were filtered out solely because of only one criterion and figure out if that was really the reason for why the query failed. Then it is possible to derive the probability for each criterion to be the reason for a query to fail. Instead of always filtering out the queries that match at least one criterion, a probability could be set for the query to be faulty because of one criterion, or a set of criteria if the query would have been filtered out because of multiple reasons. While generating the report, a limit could be set for what the probability should at least be for a query to have failed due to a known reason before filtering the event out. The probability would be a discrete variable with only a limited number of possible values due to a small number of criteria used in filtering.

Even with the current setup, there are more than enough of error events to go through manually. Therefore, putting extra effort into filtering less events out is somewhat questionable. Although, keeping more events for manual checking would increase the heterogeneity of the events. For instance, now events with certain combinations of traverse modes are filtered out because they are estimated to have failed due to no stops being near origin or destination. Hence, possible issues specific to certain combinations of modes cannot be found through this method.

There is clearly room for improvement when it comes to the tool. The current level of filtering is helpful but not enough. There are many ways to tackle this issue. For instance, the arbitrary minimum distance limit between origin and destination to get rid of events caused by points being too close to each other could be increased, or alternatively information could be fetched from OTP that this was indeed the cause for the failure. The number of events left after filtering that have failed due to the distance from a coordinate being over the maximum walking or cycling distance away from the nearest stop could be lessened. The polygon area, or areas, used to filter out events cover large areas with no stops nearby. A geographical information systems (GIS) software could be used to limit the area so that there is always a stop closer than x meters away. However, the difficulty is that the maximum walking or cycling distance can be configured to be different in each request. Although, the number of different values for this variable in the queries left after filtering was only nine. Therefore, it is reasonably feasible to maintain, or generate at runtime, a set of polygons for this purpose. Another potential issue is that this would cause areas to consist of large number of polygons which would slow down the point-in-polygon calculations done to figure out if an event should be filtered out.

From the small sample events that were manually examined, it became clear that there are events which cannot be reproduced through querying itineraries from OTP with the stored parameters. More research should be done into figuring out the cause for this. There are more than one potential root cause. Some of these events could have been triggered because the OTP instance that was handling the query was under heavy load and the search timeouted.

OTP does return information on if the itinerary search was stopped due to a timeout limit. However, these events should not be filtered out for this reason as queries that take long to process provide useful information for improving the performance. Instead, maybe these queries should be grouped for inspection.

Another reason to why an issue could not reproduced in routing is that the data was potentially different. Static data from the date an event was generated on could be used to test the reproducibility of an event, if the goal was to pinpoint the reason for why the issue could not be reproduced with the data from a future date. However, it is nearly impossible to test with the realtime data that was in use at that moment because there can be many updates even within a second and it is not known which of those the routing instance has successfully received.

The routing parameters used in attempts to reproduce the events did not necessarily match the parameters used during the original search. More parameters should be passed from OTP in the context information, and those should also be taken into use. For instance, information regarding possible intermediate places was not stored in the filtered events and it is already known that there are bugs in such itinerary searches.

The grouping of coordinates did not yet prove to be helpful. This is partly due to the sample size of events but also because not enough effort was put into optimizing it. For instance, there is no clear reason why we origins and destination should be separated for clustering. It should not matter if the coordinate is an origin or a destination in debugging potential issues in OSM network as there are only minor differences when it comes to the traverse direction in the way OTP interprets OSM data. Separate clusters were created from each routing instance type as the availability of transit data is different in those types and lack of data can cause queries to fail. However, the OSM data should be the same for the coordinates from different routing instance types as coordinates outside of the OSM area in use are filtered out. Therefore, clustering with combined coordinates from all routing instance types could be attempted.

In theory, Sentry could be used more for reactive problem solving through usage of its API together with some scripting. One of the core features of Sentry is to alert when there are new issue types. However, if a new issue, that causes no itineraries to be found, is grouped under an existing issue type, it does not raise a red flag unless it affects the total number of errors for that issue type dramatically. If errors within an error type can be categorized in an external component, an alert can be sent from there to developers to take a closer look at a potentially new or trending issue. Another option would be to use more exception types in OTP so the events would automatically be put into different issue types.

6.2 Quality benchmarking

The structure of the data in Matomo is not universal to journey planner UIs, and to successfully fetch data from Matomo with the scripts created in the Digitransit project, one would have to either use the Digitransit UI [47], or copy its behavior for use of Matomo events and Uniform Resource Locator (URL) query strings. This is not ideal and makes copying the work documented in this thesis difficult for others, but this work can be used as an inspiration for fetching coordinates from a similar data source.

The usage of OTPQA as part of the development process, which was outlined in Section 5.2.2, has been found to be useful. However, it is slightly inconvenient to use because it required manual work and time. It is only used when there is not enough confidence otherwise that the quality routing is at the expected level. Ideally, this process could be more automated. If it was more convenient to use, it would be used every time changes are done, and unintended changes to the quality of routing could be prevented with more certainty.

In theory, after some modifications, OTPQA comparison feature could be integrated into a continuous integration (CI) tool used for checking pull requests done to OTP repositories in Github, or as a separate process that gets triggered by a new pull request. Then it could be observed how a pull request would change routing's quality in terms of the available metrics. However, there are some challenges attached to it. First, the performance of the machine where the profiling is done should be stable – machines used by CI platforms are often not stable enough. This problem could be circumvented by doing the profiling and comparison on a remote machine as a separate process. Additionally, each routing instance should be tested with relatively up-to-date data that includes data types that should be supported by the instance at hand.

For the second example case described in Section 5.2.3, the reason for picking a test date one week in the future from the current date, in data loading tests, was to a get a confirmation that the routing still works with the data then. This does effectively stop new graph, which is extensively broken, from being deployed in most cases. However, there is a potential risk that the data is valid in the future, but not for the next 6 days following the build. Additionally, if the routing would work perfectly fine for the next 6 days but not for the days after that, the updated graph is not deployed. This potentially blocks important updates for the near future from being deployed to production use. An alternative method for testing could be to first test as it is done now, and if the tests fail, send an alert about it and rerun tests by subtracting one day from the date until tests pass or a hard limit, for how many days the data should be valid for, is met.

There have been occasions where the tests pass during the graph build even though there are extensive changes done to OSM data since the last successful build. For instance, a large part of the street network could be removed, and

OTP would still be able to find itineraries, but they would be using a more limited set of streets which affects the itineraries significantly. Hence, it could make sense to use the approach of comparing the new version of the router against a benchmark version instead of the more limited comparison against a static limit for success rate. Then a deployment of a new graph could be prevented if the returned itineraries differ too much from the ones from the version used as the benchmark. If larger changes are anticipated and acknowledged, thresholds for an acceptable change can be fine-tuned for that build. Problem with this approach would be finding optimal limits for how much the itineraries can differ and what characteristics are monitored.

A limit for how much the change can be to "worse" direction before the new version is considered to have changed "too significantly" was already added for each metric. This feature has not been found to be useful as of yet because the comparison is used to get more insight into changes, and for an approach that resembles approval testing, not to potentially stop some test pipeline because of the comparison has found the new version to be "worse". Most of the metrics can change to either direction and be viewed as an acceptable change depending on what was intended to change, and what side effects are acceptable. However, if use of this benchmarking method as part of the graph building process is developed, this can be used as the starting point.

In the third example of OTPQA's usage, which was introduced in Section 5.2.4, a walk reluctance parameter's value was changed, traverse mode preferences were removed and extra costs added for boarding transit vehicles were set to zero. The results were visualized on a Cartesian coordinate plane.

When other parameters were unchanged, or traverse mode preferences were removed, increasing the value of the walk reluctance parameter reduced the duration of itineraries with all the values that were tried in this study compared against a version of OTP that uses the current value. However, it seems like the duration of the itineraries would increase if the value of the walk reluctance parameter is increased too much. In theory, the duration of an itinerary in a itinerary suggestion should have the lowest possible value when the cost of traversing edges equals the time spent traversing those because then a path, which takes longer to travel on than on an alternative path, should not be used. The hypothesis is that increasing the reluctance parameter's value, which is already over one, reduces the duration of the itineraries because it counteracts the effects from use of other reluctance parameters and static costs attached to events. Evidence towards this theory could be witnessed when extra cost for boarding vehicles was set to 0 and then the duration of itineraries seemingly reaches the lowest value below the current value of the walk reluctance parameter. It is also possible that there are issues in routing which cause it to behave sub-optimally.

The increase in number of trips used in itineraries when walk reluctance parameter was increased was unsurprising. Naturally, more vehicles will be boarded on average to travel between two points if walking is avoided. The

number of trips used in itineraries was highest when the extra vehicle boarding cost was removed as there are less costs left related to public transportation to balance out the effects from the walk reluctance.

When the extra cost for boarding vehicles was removed, itineraries had shorter durations than the other two tested base configuration. This was somewhat expected as the added cost for boarding vehicles is relatively large and traversing on transit is faster than walking. The most interesting finding from this study was that the traverse mode preferences do not increase the duration of itineraries considerably with most of the tested values. On the contrary, when walk reluctance value was between 2.25 and 2.75, the itineraries had less duration with the preferences compared against no preferences. The goal behind the traverse mode preferences is to steer passengers into using train and metro instead of especially buses for travelling, as metro and train are able to handle more capacity efficiently.

In this small experiment, only few parameters were touched out of large number of available parameters that affect routing. However, it would be possible to try changing more parameters separately or simultaneously. Also, the focus was now on how reconfiguration affects the duration of the travel and the number of itinerary boardings. However, more variables could have been examined and a Pareto set, where the values are "optimal", could have been searched.

The current problem in optimizing parameters, or code, in a way that affects routing results is the lack of understanding on what an ideal itinerary suggestion should look like in general. Experts might have a strong opinion that certain itinerary suggestion is absolutely the best for travelling between points A and B, but they might not be aware of all the available options and the preferences of passengers are subjective. The current models for programmatically rating itinerary suggestions are really simple compared to the complex preferences of journey planner users.

What can be done with the parameter optimization of a routing system that uses a single criterion—cost—is to find sets of parameters that can be used as profiles. For example, if a user wants the fastest itineraries that the router is able to give, a set of parameters found to be the most optimal for that purpose can be used. Although, what is the most optimal set of parameters depends also on the data that the router uses as, for example, the structure of the transportation network can affect what is the optimal set [5].

In this thesis, the examples of how OTPQA can be used exist as a groundwork for future research. To get a better understanding on the usefulness of these approaches, a study should be conducted where the use of these methods is tracked. For example, it would be possible to track how much time is used by developers for QA, and how often potential issues are found through the usage of these methods. Similarly, it could be studied how many times OTPQA, or a similar implementation, is used as part of graph building process, how many times it stops a new graph from being deployed due to actual issues or false

positives, and how many times a faulty graph is deployed because the tests failed to notice an issue.

There is a new project under development for comparing different versions of a router, or different routing systems. Thomas Gran from the OTP developer community started this project under the name of TrakPi [78]. It is designed to have a better structure and more flexibility than OTPQA. It uses a concept of key performance indicators (KPI) to track performance of a system based on predefined criteria [78]. It is likely that at some point in the future this will replace the use of OTPQA.

6.3 Future prospects for data validation

Many of the problems in routing are due to data related issues. Not all OSM edits are useful. Quite often clearly unwanted edits are done to OSM. For example, someone can modify a street name to be a curse word. It can take a while before someone else spots it and reverts the change. Sometimes issues are not that clear-cut. A way can be added to OSM with a slightly wrong tag that causes issues in systems that consume the data. If it is known that a certain routing query fails due to improper tag usage in an OSM object, that object can be edited. However, ideally, the potential data issues should be spotted before they start affecting routing results in production. Therefore, it is important to have as many data validation steps before new data goes into real use as possible without making the process too slow, complex and hard to maintain.

At the Digitransit project, we do not have a local copy to which we would hand-pick edits from the master data. Instead, the local copy is replaced with a cut of the master data for the desired area, if OTP can load the new version without major issues and the routing tests pass. Validation of every edit that is done to OSM is a time-consuming process that requires experts who are solely focused on it, if it is done manually. Therefore, it is not a feasible option for smaller projects.

Facebook revealed that they are using machine learning to validate OSM edits that they take into their local copy of OSM [21]. If this is turns out to be successful in the long run, and more automation is added, it could make it possible for smaller scale operations to effectively maintain a local copy of OSM. Similarly, the GTFS validation tools can be improved to avoid regressions in data.

Chapter 7

Conclusions

Testing and quality assurance (QA) of a public transportation routing system is not a widely researched or documented subject. Many of the principles from general testing and QA theory apply to this domain. However, how they should be applied in this context is mostly learnt through experience. Therefore, there should be more information available on this subject so that the efforts of testing and QA could be focused to be more effective.

In this study, it was presented how known failed routing queries can be programmatically preprocessed to aide reactive identification of problems in routing. Additionally, use of benchmarking to assure quality during development, data building and reconfiguration was studied.

The use of failed routing requests for tracking issues in routing has potential, but the implementation presented in this thesis is not effective enough for filtering out enough requests caused by known issues, and it currently removes requests that should be left in for further examination. Only one request from a sample of examined requests (N=30) could be ruled to have failed with a high probability due to a potentially unknown issue. The use of benchmarking for QA in routing has been found to be useful and as an important finding, a preference of certain public transportation modes in the Helsinki metropolitan area did not seem to cause the routing to find itineraries with considerably longer durations or more transfers compared to use of no mode preferences.

More research should be done on the topic of how to measure quality of itineraries. Currently, human expertise is required in judging if an itinerary suggestion is considered to be "good" given the preconditions. The quality benchmarking tool—OTPQA—described in this study provides a method to compare a version of a routing system against the best version known of that system. What is the best version still needs to be validated by a person or group of people. The information gathered from the comparison can be helpful in deciding if a new version is better than the other versions.

There are promising advances in the field of data validation. Use of machine learning could lessen the errors in routing, as many of the problems are because of issues in data. However, it can still take a while before machine learning is taken into use smaller projects.

This thesis is focused on the use of OpenTripPlanner (OTP). However, the concepts and ideas presented here can be applied to other public transportation routing systems. One part of the quality assurance process could be to compare different routing systems against each other to get an idea on which parts some other system is performing better.

Appendix A

First appendix

A.1 Configuration files

router-config.json

```
{
  "modeWeight": {
    "BUS": 1.2,
    "SUBWAY": 0.9,
    "RAIL": 0.95
  },
  "routingDefaults": {
    "walkSpeed": 1.3,
    "transferSlack": 120,
    "maxTransfers": 4,
    "waitReluctance": 0.95,
    "waitAtBeginningFactor": 0.7,
    "walkReluctance": 1.75,
    "stairsReluctance": 1.65,
    "walkBoardCost": 540,
    "walkOnStreetReluctance": 1.5,
    "carParkCarLegWeight": 2,
    "itineraryFiltering": 2
  },
  "routePreferenceSettings": "HSL",
  "updaters": [
  ]
}
```

build-config.json

```
{
  "areaVisibility": true,
  "staticParkAndRide": false,
  "parentStopLinking": true,
  "subwayAccessTime": 0,
  "osmWayPropertySet": "finland",
  "fares": "HSL",
  "elevationUnitMultiplier": 0.1,
  "vertexConnector": "HSL"
}
```

Bibliography

[1] Aberdour, M. Achieving quality in open-source software. *IEEE software 24*, 1 (2007), 58–64.

[2] Antrim, A., and Barbeau, S. J. The many uses of gtfs data–opening the door to transit and multimodal applications. *Location-Aware Information Systems Laboratory at the University of South Florida 4* (2013).

[3] Bache, G., and Rössler, J. Approval testing: Agile testing that scales. `https://www.methodsandtools.com/archive/approvaltest.php`. [accessed. December 14, 2019].

[4] Barrett, C., Bisset, K., Holzer, M., Konjevod, G., Marathe, M., and Wagner, D. Engineering label-constrained shortest-path algorithms. In *International conference on algorithmic applications in management* (2008), Springer, pp. 27–37.

[5] Bast, H., Delling, D., Goldberg, A., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., and Werneck, R. F. Route planning in transportation networks. In *Algorithm engineering*. Springer, 2016, pp. 19–80.

[6] Ciepłuch, B., Jacob, R., Mooney, P., and Winstanley, A. C. Comparison of the accuracy of openstreetmap for ireland with google maps and bing maps. In *Proceedings of the Ninth International Symposium on Spatial Accuracy Assessment in Natural Resuorces and Enviromental Sciences 20-23rd July 2010* (2010), University of Leicester, p. 337.

[7] Delling, D., Dibbelt, J., and Pajor, T. Fast and exact public transit routing with restricted pareto sets. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)* (2019), SIAM, pp. 54–65.

[8] Delling, D., Dibbelt, J., Pajor, T., Wagner, D., and Werneck, R. F. Computing multimodal journeys in practice. In *International Symposium on Experimental Algorithms* (2013), Springer, pp. 260–271.

[9] Delling, D., Pajor, T., and Werneck, R. F. Round-based public transit routing. *Transportation Science 49*, 3 (2014), 591–604.

[10] Delling, D., Sanders, P., Schultes, D., and Wagner, D. Engineering route planning algorithms. In *Algorithmics of large and complex networks*. Springer, 2009, pp. 117–139.

[11] Digitransit. Digitransit the next generation journey planner. `https://digitransit.fi/en/`. [accessed. December 13, 2019].

[12] Fewster, M., and Graham, D. *Software test automation*. Addison-Wesley Reading, 1999.

[13] Hart, P. E., Nilsson, N. J., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics 4*, 2 (1968), 100–107.

[14] JUnit. Junit4 about. `https://junit.org/junit4/`. [accessed. November 22, 2019].

[15] Knowles, N. The google transit feed specification – capabilities & limitations a short analysis. `http://www.normes-donnees-tc.org/wp-content/uploads/2014/05/GoogleTransitReview-05.doc`, Jul 2007. [accessed. April 30, 2019].

[16] Liebig, T., Piatkowski, N., Bockermann, C., and Morik, K. Route planning with real-time traffic predictions. In *Proceedings of the 16th LWA* (2014), pp. 83–94.

[17] Matomo. Why matomo? `https://matomo.org/why-matomo/`. [accessed. September 30, 2019].

[18] McHugh, B. Pioneering open data standards: The gtfs story. *Beyond transparency: open data and the future of civic innovation* (2013), 125–135.

[19] Mellemstrand, G., and Gran, T. Opentripplanner current features and the motivation for 2.0. `https://docs.google.com/presentation/d/1YbHl6syZy6n63wiibDKUp9WECb9K8BpH2orHexs7xXQ/edit`. [accessed. December 1, 2019].

[20] MobilityData. Gtfs-flex. `https://github.com/MobilityData/gtfs-flex`, Jan 2019. [accessed. April 30, 2019].

[21] Mohapatra, S. Mars: How facebook keeps maps current and accurate. `https://engineering.fb.com/ml-applications/mars/`. [accessed. November 27, 2019].

[22] Neis, P., Zielstra, D., and Zipf, A. The street network evolution of crowdsourced maps: Openstreetmap in germany 2007–2011. *Future Internet 4*, 1 (2012), 1–21.

[23] Orda, A., and Rom, R. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM (JACM) 37*, 3 (1990), 607–625.

[24] Pyrga, E., Schulz, F., Wagner, D., and Zaroliagis, C. D. Experimental comparison of shortest path approaches for timetable information. In *ALENEX/ANALC* (2004), Citeseer, pp. 88–99.

[25] QGIS. Qgis a free and open source geographic information system. `https://www.qgis.org/en/site/`. [accessed. September 30, 2019].

[26] Apache Software Foundation. Maven surefire plugin. `https://maven.apache.org/surefire/maven-surefire-plugin/index.html`. [accessed. November 22, 2019].

[27] Apple, I. Apple maps. `https://www.apple.com/ios/maps/`. [accessed. December 13, 2019].

[28] CGI, I. Navici trip planner. `https://www.cgi.com/sites/default/files/brochures/navici-trip-planner.pdf`. [accessed. December 13, 2019].

[29] European Commission. Commission delegated regulation (eu) 2017/1926. *Official Journal of the European Union* (2017).

[30] Functional Software, I. About sentry. `https://sentry.io/about/`. [accessed. September 30, 2019].

[31] Functional Software, I. Grouping & fingerprints. `https://docs.sentry.io/data-management/event-grouping/`. [accessed. September 30, 2019].

[32] Functional Software, I. Understand the entire context. `https://sentry.io/features/context/`. [accessed. September 30, 2019].

[33] Google, I. Google maps. `https://www.google.com/maps/`. [accessed. December 13, 2019].

[34] Google, I. Gtfs realtime overview. `https://developers.google.com/transit/gtfs-realtime/`, Oct 2018. [accessed. May 1, 2019].

[35] Google, I. Revision history. `https://developers.google.com/transit/gtfs/guides/revision-history`, Oct 2018. [accessed. April 30, 2019].

[36] Google, I. Service alerts. `https://developers.google.com/transit/gtfs-realtime/guides/service-alerts`, Oct 2018. [accessed. May 1, 2019].

[37] Google, I. Trip updates. `https://developers.google.com/transit/gtfs-realtime/guides/trip-updates`, Oct 2018. [accessed. May 1, 2019].

[38] Google, I. Vehicle positions. `https://developers.google.com/transit/gtfs-realtime/guides/vehicle-positions`, Oct 2018. [accessed. May 1, 2019].

[39] Google, I. General transit feed specification reference. `https://developers.google.com/transit/gtfs/reference/`, Apr 2019. [accessed. April 30, 2019].

[40] Google, I. Google transit extensions to gtfs. https://developers.google.com/transit/gtfs/reference/gtfs-extensions, Apr 2019. [accessed. April 30, 2019].

[41] Google, I. Revision history. https://developers.google.com/transit/gtfs-realtime/guides/revision-history, Apr 2019. [accessed. May 1, 2019].

[42] GraphHopper GmbH. An open source route planning library and server using openstreetmap. https://github.com/graphhopper/graphhopper. [accessed. December 13, 2019].

[43] GraphQL Foundation. Introduction to graphql. https://graphql.org/learn/. [accessed. November 24, 2019].

[44] HaCon Ingenieurgesellschaft mbH. Trip planner and travel companion. https://www.hacon.de/en/solutions/trip-planner-and-travel-companion/. [accessed. December 13, 2019].

[45] Helsinki Regional Transport. Cycling and walking. https://www.hsl.fi/en/information/sustainable-modes-transport/cycling-and-walking. [accessed. May 1, 2019].

[46] Helsinki Regional Transport. Open data. https://www.hsl.fi/en/opendata. [accessed. December 11, 2019].

[47] Helsinki Regional Transport Developer Community. Digitransit-ui is a mobile friendly user interface built to work with digitransit platform. https://github.com/hsldevcom/digitransit-ui. [accessed. December 3, 2019].

[48] Helsinki Regional Transport Developer Community. Localroute.js. https://github.com/HSLdevcom/localroute. [accessed. December 13, 2019].

[49] Infotripla Oy. Infotripla oulu siri services user documentation. http://wp.oulunliikenne.fi/wordpress/wp-content/uploads/2014/08/Infotripla_SIRI_Oulu_documentation_ver1.pdf, Aug 2014. [accessed. May 2, 2019].

[50] Mapbox, I. Validating openstreetmap. https://labs.mapbox.com/mapping/validating-osm/. [accessed. December 11, 2019].

[51] Microsoft, I. Bing maps. https://www.bing.com/maps. [accessed. December 13, 2019].

[52] National Land Survey of Finland. Elevation model 10 m. https://www.maanmittauslaitos.fi/en/maps-and-spatial-data/

expert-users/product-descriptions/elevation-model-10-m. [accessed. May 1, 2019].

[53] National Land Survey of Finland. Elevation model 2 m. https://www.maanmittauslaitos.fi/en/maps-and-spatial-data/expert-users/product-descriptions/elevation-model-2-m. [accessed. May 1, 2019].

[54] National Land Survey of Finland. National land survey open data attribution cc 4.0 licence. https://www.maanmittauslaitos.fi/en/opendata-licence-cc40. [accessed. December 31, 2019].

[55] North American Bike Share Association. General bikeshare feed specification. https://github.com/NABSA/gbfs, Feb 2018. [accessed. May 1, 2019].

[56] North American Bike Share Association. Systems. https://github.com/NABSA/gbfs/blob/master/systems.csv, Apr 2019. [accessed. May 1, 2019].

[57] Open Data Commons. Open database license (odbl) v1.0. https://www.opendatacommons.org/licenses/odbl/1.0/. [accessed. April 30, 2019].

[58] OpenStreetMap contributors. Copyright and license. https://www.openstreetmap.org/copyright/en. [accessed. April 30, 2019].

[59] OpenStreetMap contributors. Digitransit. https://wiki.openstreetmap.org/wiki/Digitransit. [accessed. November 25, 2019].

[60] OpenStreetMap contributors. Import. https://wiki.openstreetmap.org/wiki/Import. [accessed. December 11, 2019].

[61] OpenStreetMap contributors. Opentripplanner. https://wiki.openstreetmap.org/wiki/OpenTripPlanner. [accessed. November 25, 2019].

[62] OpenStreetMap contributors. Quality assurance. https://wiki.openstreetmap.org/wiki/Quality_assurance. [accessed. December 11, 2019].

[63] OpenStreetMap contributors. Tasking manager/validating data. https://wiki.openstreetmap.org/wiki/Tasking_Manager/Validating_data. [accessed. December 11, 2019].

[64] OpenStreetMap contributors. Elements. https://wiki.openstreetmap.org/wiki/Elements, Jun 2017. [accessed. May 1, 2019].

[65] OpenStreetMap contributors. Relations/proposed/site. https://wiki.openstreetmap.org/wiki/Relations/Proposed/Site, Oct 2018. [accessed. May 1, 2019].

[66] OpenTripPlanner contributors. Basic otp architecture. http://docs.opentripplanner.org/en/dev-1.x/Architecture/. [accessed. December 7, 2019].

[67] OpenTripPlanner contributors. Configuring opentripplanner. http://docs.opentripplanner.org/en/dev-1.x/Configuration/. [accessed. December 3, 2019].

[68] OpenTripPlanner contributors. Getting opentripplanner. http://docs.opentripplanner.org/en/dev-1.x/Getting-OTP/. [accessed. December 7, 2019].

[69] OpenTripPlanner contributors. Gnu general public license. https://github.com/HSLdevcom/OTPQA/blob/master/COPYING. [accessed. December 3, 2019].

[70] OpenTripPlanner contributors. An open source multi-modal trip planner. https://github.com/opentripplanner/opentripplanner. [accessed. December 13, 2019].

[71] OpenTripPlanner contributors. Opentripplanner. http://docs.opentripplanner.org/en/dev-1.x/. [accessed. December 7, 2019].

[72] OpenTripPlanner contributors. Opentripplanner basic tutorial. http://docs.opentripplanner.org/en/dev-1.x/Basic-Tutorial/. [accessed. December 7, 2019].

[73] OpenTripPlanner contributors. Opentripplanner project history. http://docs.opentripplanner.org/en/dev-1.x/History/. [accessed. December 7, 2019].

[74] OpenTripPlanner contributors. Otpqa. https://github.com/opentripplanner/OTPQA. [accessed. December 3, 2019].

[75] OpenTripPlanner contributors. Project governance. http://docs.opentripplanner.org/en/dev-1.x/Governance/. [accessed. December 7, 2019].

[76] OpenTripPlanner contributors. Routing bibliography. http://docs.opentripplanner.org/en/dev-1.x/Bibliography/. [accessed. December 1, 2019].

[77] OpenTripPlanner contributors. Security. http://docs.opentripplanner.org/en/dev-1.x/Security/. [accessed. December 7, 2019].

[78] OpenTripPlanner contributors. Transit routing assurance using performance indicators. https://github.com/opentripplanner/TrakPi. [accessed. December 17, 2019].

[79] OpenTripPlanner contributors. Troubleshooting routing. http://docs.opentripplanner.org/en/dev-1.x/Troubleshooting-Routing/. [accessed. December 7, 2019].

[80] Software Testing Magazine. Approval testing. https://www.softwaretestingmagazine.com/knowledge/approval-testing/. [accessed. December 14, 2019].

[81] United States Access Board. Chapter 4: Ramps and curb ramps. https://www.access-board.gov/guidelines-and-standards/buildings-and-sites/about-the-ada-standards/guide-to-the-ada-standards/chapter-4-ramps-and-curb-ramps. [accessed. December 6, 2019].

[82] Ritter, N., and Ruth, M. Geotiff format specification geotiff revision 1.0. *SPOT Image Corp* (2000).

[83] Tobler, W. Three presentations on geographical analysis and modeling. Tech. rep., National Center for Geographic Information and Analysis, 02 1993.

[84] Transmodel. Implementations. http://www.transmodel-cen.eu/category/implementations/. [accessed. April 30, 2019].

[85] Tryti, A. Opentripplanner in norway. https://www.entur.org/wp-content/uploads/2019/04/OTPSummit-2019-OTP_in_Norway_and_EU_regulations-Andreas-Tryti.pdf. [accessed. April 30, 2019].

[86] Turku. Tsjl - siri. https://data.foli.fi/doc/siri/v0/index-en. [accessed. May 2, 2019].

[87] Vanhanen, K., Hastrup, T., Huotari, M., and Vuorio, M. Digitransit - uuden sukupolven reittiopas leviää uusille alueille. https://paikallisliikenneliitto.fi/wp-content/uploads/2018/01/9T_Vanhanen_Digitransit_uuden_sukupolven_reittiopas_uusille_alueille_20170921.pdf, Sep 2017. [accessed. April 30, 2019].