BACHELOR THESIS

Jakub Matěna

Searching public transport connections based on transport timetables and individual transport

Department of Software Engineering

Supervisor of the bachelor thesis: doc. Mgr. Martin Nečaský, Ph.D.
Study programme: Informatics
Study branch: Programming and SW systems

Prague 2019
I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date 16. 5. 2019 signature of the author
Thanks to:

Supervisor of this thesis who helped me to find an interesting topic and acquire the open data.

My grandfather who did the language correction of this text.
Title: Searching public transport connections based on transport timetables and individual transport

Author: Jakub Matěna

Department: Department of Software Engineering

Supervisor: doc. Mgr. Martin Nečaský, Ph.D., Department of Software Engineering

Abstract: This thesis deals with routing over open data in Prague. Its main purpose is to create server/client based application that makes connection searches combining both public transport and individual car transport. Server provides Protocol buffers API that can be used by anyone to create client application for any available platform. Moreover demonstrative client application for Android devices was created as part of this project. All components were tested for usability, functionality and speed. Open data from several sources were used to demonstrate that publishing data can offer new possibilities and that we can achieve complex goals by combining them. It was shown that intermodal planning in Prague is possible and can save time, money and even environment.

Keywords: intermodal planner, open data, public transport, car transport, A*
Název práce: Hledání dopravních spojení na základě jízdních řádů a individuální dopravy

Autor: Jakub Matěna

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: doc. Mgr. Martin Nečaský, Ph.D., Katedra softwarového inženýrství


Klíčová slova: intermodální plánovač, otevřená data, veřejná doprava, autodoprava, A*
Contents

Introduction 3

1 Analysis 6
   1.1 Module analysis ........................................................................... 6
   1.2 User requirements analysis ................................................................. 6
      1.2.1 Functional requirements .............................................................. 6
      1.2.2 Nonfunctional requirements ......................................................... 7
   1.3 Open data ...................................................................................... 7
      1.3.1 Open data used ............................................................................. 7
      1.3.2 Data availability problems .............................................................. 10
      1.3.3 Another useful data ...................................................................... 11
   1.4 Related work ................................................................................. 11
      1.4.1 In Prague .................................................................................... 11
      1.4.2 In the Czech Republic ................................................................. 14
      1.4.3 In the World ................................................................................. 16

2 Design 18
   2.1 Module scheme .............................................................................. 18
      2.1.1 Server ....................................................................................... 18
      2.1.2 Client ....................................................................................... 19
   2.2 Choosing programming language ........................................................ 19
      2.2.1 Server ....................................................................................... 19
      2.2.2 Client ....................................................................................... 19
   2.3 Data handling ................................................................................. 20
      2.3.1 Data access .................................................................................. 20
      2.3.2 Analyzing data .......................................................................... 21
      2.3.3 Handling data format changes ...................................................... 22
      2.3.4 Parsing data ................................................................................ 22
   2.4 Routing in public transport ................................................................. 23
      2.4.1 Routing algorithms ..................................................................... 23
      2.4.2 Using delays ............................................................................... 25
      2.4.3 Handling transfers ...................................................................... 25
   2.5 Car and pedestrian routes ................................................................. 25
      2.5.1 Custom-created solution .............................................................. 25
      2.5.2 Online providers ....................................................................... 25
      2.5.3 Running own routing server ......................................................... 26
      2.5.4 Comparison of providers ............................................................. 26
   2.6 Combining car and public transport routes ........................................... 26
   2.7 Different OS on client devices ............................................................ 27

3 User documentation 28
   3.1 Installing via APK .......................................................................... 28
   3.2 Initial start up .................................................................................. 28
      3.2.1 Graphical user interface ............................................................... 28
   3.3 Basic usage .................................................................................... 28
Introduction

There are a lot of people living outside of Prague who travel to Prague every day. It is because of work, school and many other reasons. For some of them traveling by car is the best option. But where do they park the car? There are a lot of parking places all over Prague, but which are empty and close to some public transport stop and well connected to the destination?

Organization standing behind Prague’s public transport is called ROPID (Regional organizer of Prague’s integrated transport). They publish public transport timetables among other data sets and they even publish data about delays of some of the public transport vehicles. This data could be used to route users through the public transport in Prague.

Another important organization is Operator ICT, which is Prague-owned joint-stock company. They publish a lot of data about Prague, but most importantly data about some of the Prague’s parking lots, including real-time availability. This data could be used to find suitable parking places for users.

Almost everybody knows providers of online maps (such as Google maps or Seznam maps), that enable users to browse through maps, make searches and even find routes. Most of them provide API access to enable third-party applications to use this feature. This could be used to find the shortest/fastest route from some starting position to available parking places.

Problem definition

The goal is to create an application for mobile devices that can find fastest route between two public transport stops or between current position and some public transport stop. In the case of stop to stop connection, only public transport is used. In the other case, we use car transport to get from the current position to a free parking spot and then continue by public transport to the final destination.

For this task the application should use open data about Prague’s public transport timetables, public transport delays and available parking places. For the car transport an external service will be used to determine travel times. Open data from different sources can be combined to achieve better results than using every data set separately. As a demonstration data about air quality should be used to prefer public transport in case of poor air quality.

The quality and accessibility of this data is getting higher every day and it should be possible to create much needed software solution for this problem.

In brief

Main target platform is Android, but the application should function on server-client basis, separating two main tasks into separate problems. Client application
is written in Java using Android Studio. This means that the application runs only on Android devices, but it is only a thin client and its main function is to provide GUI and connection to server. Creating client application for other platforms should be relatively easy and should require almost no knowledge about the server application and its implementation.

The server application is responsible for solving the main task - to find the route. Its objective is to download all necessary data, combine them and find the fastest route based on requirements received from client application. Its main criteria are answering the request as fast as possible. There can also be multiple concurrent requests. Therefore, maximizing speed is one of the most important goals.

The server part is implemented in C++ using Qt library, especially its SQL access and other useful features. It is made out of two applications. The first creates database from input data and the other accepts and handles requests.

Testing was done in four different ways:

- Application was tested using small C++ application. Up to 100 parallel searches were made to test, if server side can handle multiple requests and to measure speed under heavy load.

- Client side was tested by users to find out, what is the usability of the client application. How easy it is to use it and so on.

- Data consumption was measured to find out how efficient the application is.

- And finally the application was installed and tested on different Android devices.

**Document structure**

**Analysis**

This section presents detailed problem description and comparison to other solutions that deal with identical or similar problems. It is also about open data that is or could be used.

**Design**

Design contains reasoning behind chosen solutions, possible alternatives and important advantages and disadvantages.

**User and programmer documentation**

This section contains user and programmer documentation. User documentation describes all program functions and shows how to use it. Including typical use
cases, enabling user to use the application as intended. Programmer documentation describes the code itself. Namely chosen solutions for problems encountered during implementation. Possible other solutions and hints how to improve or expand the program.

Testing
This chapter is about testing of the software and results of these tests. Specifically, about performance of the implementation and graphical user interface usability.
1. Analysis

This chapter presents detailed problem description and comparison to other solutions, that deal with identical or similar problems. It is also about open data that is or could be used.

1.1 Module analysis

The application should run on mobile devices. One option was to create only mobile application that would do all the work. Theoretically internet access would be required only for data updates and the application could be used even without data connection. Other is to use the server/client scheme and do all the work on server side.

1.2 User requirements analysis

This is a summary of user requirements and how the final product should behave and function.

1.2.1 Functional requirements

- The application should be able to find route between two public transport stops. For this purpose, only several arguments should be needed, consisting of two stop names and origin departure time and date. The connection found should be the fastest possible in the given time. The result should be based on real data describing public transport in Prague and it should be one specific connection describing times of departure, arrivals and transfers. There should be enough time for transfers between different transport vehicles. Delay times should be used to make searches more relevant. If one transport is delayed, we should presume, that we might not catch the following one. This mode is intended for people living in Prague, that travel to work or school via public transport and they want to find the fastest connection.

- It should be possible to find connection combining public transport with individual car transport. Specifically using car to get near or in Prague and then continue using public transport. The result should be based on the previous functionality and some service about car transport that can precisely calculate transport times to suitable parking lots. Data about parking lots should be used to find out empty parking places. There should be enough time to transfer between parking lot and a public transport stop. The found connection should consist of specific departure, arrival and transfer times.
1.2.2 Nonfunctional requirements

- The computing should be ideally handled on server side to solve the task faster and save battery and mobile data.
- Application should be relatively simple and immediately ready to go with no need to change preferences or settings.
- Using the application should be easy and fast to learn.
- The application should be cheap for users, without need to pay anything.
- There should not be any annoying advertisement. It only consumes mobiles resources, such as battery, processor power and mobile data.
- Mobile data is precious resource, especially in the Czech Republic, where Prague is located. Therefore, minimum amount of mobile data should be used to prevent unnecessary costs for users.

1.3 Open data

In this part, we are going to concentrate on open data, that is used or could be used to solve the problem. Mainly what information they hold and their formatting.

1.3.1 Open data used

Public transport timetables data

Main data used by the application is data about public transport timetables. Data is available on web\(^1\) at the time of writing this thesis. The data is stored in ZIP file format. After unzipping the file, we get 13 text files in GTFS format. Let’s have a look at the most important ones.

- File *stops.txt* contains information about all stops in the system. Each stop has along other parameters an ID and a name. Some stops have identical name. That is because name is designation for users and stops with the same name are usually part of one bigger whole, for example metro station, but more commonly it is just two stops for bus, one for each direction. Other parameters are position of the stop, described by latitude and longitude. There is also other information, that is currently not used in this work. For example, link to a web page about the stop or information about zone in which the stop resides. Each line of this file is a vertex in the graph, that is used for finding the route in our application.

- File *routes.txt* holds information about routes of buses, trams, metro, ferries and trains. It contains general information about the route. Property *route_short_name* is designation of route for passengers, e.g. 185, 177 (bus), 22, 9 (tram) or A, C (metro). Another property *route_type* defines if the route is bus, tram, metro, ferry or train.

\(^1\)http://data.pid.cz/PID_GTFS.zip
• File *trips.txt* contains data about each trip. Each trip is part of some route. Every trip has service ID, which refers to file *calendar.txt*. Each trip has one time of a day and several days in week, when it operates. Days of the week are specified in *calendar.txt* and times in *stop_times.txt* file.

• File *stop_times.txt* holds information about specific arrivals and departures of some trip at certain stop. Most important properties being *trip_id*, *arrival_time*, *departure_time* and *stop_id*. This is the largest file and contains most of the important data.

• File *calendar.txt* defines what *service_id* means. It defines on which days of the week the trip operates and from and to which date it operates.

• File *route_sub_agencies.txt* defines which route is managed by which agency.

The other files are not used in the route searching algorithm. They are holding additional information about public transport.

All the data is guaranteed to be valid for at least a week. Sometimes some of the trips are valid for longer as described in *calendar.txt*. But generally, there is no guarantee for longer validity times.

**Public transport transfer data**

Data about transfers is not publicly available at the moment. That means that there is no good way to find out, how long a transfer, e.g. between metro stop and bus stop, will take. There are several options how to solve this problem.

First option is to use a routing machine to find a route. In the data about timetables, there are coordinates for every stop and so it should not be a problem to use some map data and some engine to do the job. Yet problems arise in the most common scenario. Coordinates of underground stations do not take in account the underground factor and the time it takes to actually get to the surface. Therefore, when metro stop and bus stop are directly above each other and the coordinates are almost the same, it might take several extra minutes to actually get to the other stop.

But not only underground-surface transfers are a problem. Map data is not perfect and even though street and roads are very reliable, data about pavements and other viable paths for pedestrians are problematic at best. All in all, routing approach is currently not very reliable and can result in too long or too short transfer times.

Solution we chose to use is very simple and not perfect, but for most of the transfers it works very well. The standard transfer time is 120 seconds for every transfer. But it is possible to change values for certain transfers via special file. Currently I am working on another desktop program that should enable to edit transfer times quite easily through graphical interface. This way if anyone finds out some transfer time, that is just not right, he/she can change the transfer time and create patch file that can be used on server side and solve the transfer for everyone.
There is also a file called transfers.txt in data about timetables that is describing transfers in a different way. It describes transfer between two routes, where one route waits for the other and sometimes there is also information about predicted transfer time. This data is used as well.

It would be best, if someone measured the time needed to get from underground to some location on surface and then in combination with better maps, it would be possible to find reasonable transfer times to other stops. But this is not possible just now.

**Public transport delay data**

Every public transport vehicle has a GPS locator and sends regularly its position to the ROPID. Unfortunately, only locations of vehicles operated by private companies are public. Most of which are intercity buses. Nevertheless, the data is used for route searching and if in the future all data is made public, our application is ready. Data contains current position, current delay in seconds and recorded delays it had in previous stops. Our solution uses only current delay and plans the routes accordingly.

**Parking lot data**

Data about Park and Ride parking lots is used to find a suitable free parking place, when performing combined route search. Every parking lot has its identifier, total number of all places and number of currently free spaces, address, precise position given by latitude and longitude and user-friendly name. There are also other parameters, which are not used in our solution.

Currently there are only about 30 parking lots in the data. But there are much more parking lots and individual parking places all around Prague. Therefore 150 random static parking lots were generated to test the applications behavior with higher number of parking lots. The format of the generated data is very similar to the format of real data, only difference being, that parameters not used are not generated.

This work might also serve as motivation for authorities to publish data about other parking places and even create some mechanism to find out which places are free, and which are taken. One possibility is control cars, that already circle through the city and check if parked cars are legally parked. These cars could easily detect free parking spaces.

**Air quality data**

Data about air quality is supplied by ČHMU\(^2\) and are currently available at their web page\(^3\). There is data from several stations all around the Czech Republic and even some neighboring countries.

\(^2\)Czech Hydrometeorological Institute
\(^3\)http://portal.chmi.cz/files/portal/docs/uoco/web_generator/aqindex_cze.json
Our project concentrates only on Prague and its surroundings. Therefore, we use data about Prague only. For simplicity we also use only summarized data about air quality. That means that we get one value on scale from 1 to 6 about Prague center and one about Prague surroundings. Main goal, why we use the data is to show, what benefits we can achieve by combining the open data. In this case it is preference of public transport to individual, if air quality is poor.

**Open street maps data**

The application uses Open street routing machine to find shortest routes in two main scenarios. The first is individual car transport from any location to any free parking space. The second is pedestrian transfer from this parking space to some public transport stop. In both cases Open street maps are used as data for Open street routing machine.

### 1.3.2 Data availability problems

In this subsection we will summarize, which data sets, that are currently unavailable might be used to enhance the usability in the future.

**Delay data for all routes**

Currently only delay data is available for routes operated by other agencies than the main one - Prague Public Transit Co. Inc. [DPP]. This is mainly because of disadvantageous contract between this agency and their provider of GPS locators in buses and trams. They are not allowed to use the location data as they please and therefore they cannot publish it. And because delay time calculations are based on the current locations of the transport vehicles, the delay data is also not available. Only upside to all this is the fact that most of the routes operated by [DPP] are quite reliable and therefore missing delay data is not such a big problem. Also, the contract should expire in a few years and then it might be possible to publish the data.

**Transfer data**

Other data that would be very useful is data about transfer times between individual stops. Mainly between stops with the same name like metro stations or bus stations, where transfers are widely used. Calculating transfer time based on stop’s location and map data is very unreliable.

**More data about parking spaces**

Possible expansion of data about parking places would enable to find even better routes while combining individual and public transport. There are several approaches, how to collect the data.

Data from parking machines in paid zones might be used to find out how many empty places are still in the zone. Unfortunately, not all parked cars must have a ticket. E.g. having a special permission or parking without any permission at all.
Other source might be special cars circling through the paid zones and using cameras to identify, if all the parked cars have permission to park there. They can easily spot an empty parking place. Problem is that they are using some special pattern and the authorities are afraid that publishing the data might reveal the pattern and help drivers to avoid being detected, when they are parking, where they should not.

Last idea is to install special cameras or some kind of sensors to detect cars passing through some street. If there were a sensor at the beginning and at the end of the street, we could easily calculate number of cars staying in the area. This together with number of available parking places in the area gives us number of not yet taken parking places.

1.3.3 Another useful data

There is also data, that might be available and could be useful, but is currently not used. Mainly because of limited complexity of this work.

Bike sharing

Data from bike sharing services might be used to allow users to use bike sharing as another means of transport. If specific coordinates of all the available bikes were known, it should not be a problem to use bikes instead of waiting for bus or tram. User could also specifically set to prefer bikes to buses.

1.4 Related work

In this section we present some of the solution, that have already been created for similar problems. Mainly route finding in public transport.

1.4.1 In Prague

Dpp.cz

On web page of Prague Public Transit Co. Inc. (DPP), it is possible to search a connection above Prague’s public transport data only. After making the search, multiple results are shown as you can see in CHAPS Web design of the page was created by WDF - Web Design Factory and the back end was created by company CHAPS.

Fuzee

This application is combining Prague’s public transport and bike-sharing services. It is possible to search connection that uses public transport, walking and cycling. The application is distributed as mobile app and the results are shown on a map as you can see in More information can be found on Fuzee web page.
Connection: Vysočanská - Chodov

3. 5. 2019 8:35:00

- 8:35 - 9:25, 50 min.
  Vysočanská, 8:35 → Metro → Chodov, 9:25
  Price: 32 CZK, for information on tariff zones, click on the connection detail.
  Traveling time: 50 min., distance: 10 km.
  Runs on 🚄.

3. 5. 2019 8:38:00

- 8:38 - 9:07, 29 min.
  Vysočanská, 8:38 → Bučický → Florenc, 8:46
  Můžete jít 3 minut, Move about 3 minutes
  Florenc, 8:51 → Florenc → Chodov, 9:07
  Price: 24 CZK, for information on tariff zones, click on the connection detail.
  Traveling time: 29 min., distance: 14 km.
  Runs on 🚄.

3. 5. 2019 8:40:00

- 8:40 - 9:07, 27 min.
  Praha Vysočany, 8:40 → Metro → Chodov, 9:07
  Můžete jít 6 minut, Move about 6 minutes
  Chodov, 8:52 → Chodov → Chodov, 9:07
  Price: 24 CZK, for information on tariff zones, click on the connection detail.
  Traveling time: 27 min., distance: 14 km.
  Runs on 🚄, runs not 8.5.
Figure 1.2: Fuzee results.
In the Czech Republic

IDOS

IDOS is a public transport route searching application. It uses public transport timetables of intercity buses, trains and also city transport. In offers additional information about the routes and in some cases also allows user to buy or book a ticket for chosen trip as can be seen in Figure 1.3. Company CHAPS is responsible for the technological solution and maintaining the data.

Pubtran

This application is developed by internet company Seznam.cz. It is using data published by company CHAPS. Since year 2015 this data is published in machine readable format. This makes it possible for Seznam.cz to use the data. The application offers search of routes using public transport in combination with pedestrian transport, the result can be seen in Figure 1.4. It can also be easily shown on a map using map service Mapy.cz, that is also owned by Seznam.cz.

Figure 1.3: IDOS results.
Figure 1.4: Pubtran results.
Mapy.cz

Web page Mapy.cz enables users to browse through world maps and find routes using either car, bike, walking or combination of public transport and walking. It can also search cross-country skiing and canoeing routes. The results are displayed on the map 1.5.

CG transit

This mobile application enables to find routes using public transport. The app is free. But it is necessary to pay a yearly fee to acquire data that the application uses. One of its advantages is the possibility to search even without internet connection. It supports Android devices, iPhone and iPad.

1.4.3 In the World

Google Maps

Google maps7 offer similar services as Mapy.cz. They also provide car, bike, walking and combination of public transport and walking searches. In addition, they can also provide flight searches. The results are again displayed on the map.

Helsinki Region Transport

Helsinki Region Transport is joint local authority founded by Helsinki and surrounding municipalities. It was started in year 2010. The service can combine public transport with walking, cycling or driving. The results are shown on the map 1.6. More information and the service itself can be found on their web page8.

---

7 https://maps.google.com
8 https://www.hsl.fi/en
Figure 1.6: Helsinki Region Transport result.


2. Design

In this part we are going to summarize, what decisions were made and what obstacles had to be overcome. Everything from choosing appropriate programming language, acquire and parsing the data to implementing the algorithms and using already created solutions for partial problems.

2.1 Module scheme

Server/client scheme was chosen instead of mobile only application. Following are the several of the most important reasons:

- Mobile devices are not very fast and computing the task might take longer time.
- Some of the data sets are constantly changing and so the internet access would be required almost every time a request was made, therefore it would not work properly without constant internet access.
- There are multiple operating systems running on different mobile devices and the algorithm would have to be implemented for every one of them.
- Downloading the data takes some time and it can be considered recent for only limited amount of time. If the task is handled on the server side with potentially more requests, the downloaded data could be reused and the average speed of the search and even usage of internet access would be decreased.

2.1.1 Server

There are two applications on server side.

The first one called databaseCreator, which downloads data about public transport timetables and parking lots and creates the database. This database contains data about stops, public transport arrivals and departures and also transfer times between certain stops. Data about parking lots is used to create special stops for each parking lot and corresponding transfers to several closest public transport stops.

Second one is the main server application. This application uses the database created by databaseCreator to find the fastest routes above it. It also uses data about available parking places in Prague, as well as data about public transport delays and quality of air in Prague. If quality of air is poor, public transport is preferred to individual car traffic.
Server accepts requests from client application and sends back responses. Protocol buffers library is used for data transfer, making it easy to send complex data over sockets without worrying about format mismatch. This is very useful, especially when using two different programming languages for client and server. And also, if another client application should be implemented, no problems should arise.

One of the server parts is also several Shell scripts used, for tasks, that might prove complicated in C++, but are very easy in Shell. One of which is always maintaining the application running, starting it after reboot of the server, and creating the database every day as new timetables are published by ROPID.

### 2.1.2 Client

Client application is quite simple and works mainly as a means of sending the input parameters to the server. Protocol buffers are also used for communication over internet. As possible expansion more client applications for other platforms could be created, making it possible for users using other system than Android to use the application.

### 2.2 Choosing programming language

There are two parts - client and server. There is no need to use the same programming language for both.

#### 2.2.1 Server

Server does not have to be platform independent. We can choose the necessary hardware and install software needed for our server application to function properly. Main goal is to be able to achieve high speed and quickly and correctly answer request made by clients.

Server application targets Linux servers and therefore C++ was chosen. To handle database and some other tasks, that are not well supported in plain C++, therefore Qt library was used to make implementation easier. Apart from C++, some supportive tasks were handled using Shell scripts, because it is very easy to use them for some tasks, that might otherwise be relatively complicated in C++.

#### 2.2.2 Client

Client should be very easy to install. It will not do much computing and speed requirements are relatively low. Main goal is simplicity and usability. Because there are several operating systems used on mobile devices, there should ideally be implementation for each of them. However, in this work only Android implementation was created. On the side of the client only user interface is handled and therefore it is relatively easy to implement client application for other platforms.
Because we are only implementing for Android, Android Studio was chosen as IDE (Integrated development environment) and the language used is Java. It is quite easy to create a Java application for Android phones using Android Studio, because there is a built-in library providing all the necessary user interface components.

2.3 Data handling

This part is about data and what steps had to be done to acquire, analyze, understand and parse it into a fitting format.

2.3.1 Data access

A lot of data sets had to be used in the implementation of the algorithm.

1. The most importantly data about public transport timetables. This data is available in the GTFS (General Transit Feed Specification) format on web page of Prague1.

2. Secondly data about delays of public transport had to be acquired. We are accessing this data through link, that was acquired directly from ROPID2. The data was published not even year ago, so it is quite new way to expand the solution’s features.

3. These two data sets enable us to make searches in the PID (Prague’s integrated transport). But the ultimate goal is to create a solution to find combined route. This route should consist of individual car travel, parking the car, transfer from parking lot to some public transport stop and then finishing the route using only public transport and possibly some transfers between stops. Therefore, data about parking lots in Prague and some way of finding routes via car is also necessary.

4. Data about some of the parking lots in Prague is published by Operator ICT. It is mostly Park and Ride parking lots and some other private owned parking lots. Park and Ride (shortly P+R) are special parking lots on the outskirts of Prague built in order to enable people that are coming to Prague to park their cars and continue via public transport. There is only a small fee for parking there. The private parking lots in the data are mostly also paid and are usually close to some shopping center.

5. The parking data is available in the platform Golemio. Their API3 is only accessible to registered users. After creating the account, the user receives authentication data. Using the authentication data, a time limited key can

1 http://opendata.praha.eu/dataset/dpp-jizdni-rady
2 https://pid.jakluk.me/trip_updates.json
3 https://golemio.docs.apiary.io
be obtained. Finally using this key, the API can be accessed and data about parking lots can be downloaded. In fact, not only data about parking lots but also any other data sets available in the platform as well.

6. The map data of the Czech Republic is needed to find car routes from starting position to one of the parking lots. Free and public source is Open street maps, in our case [Czech Republic region](https://download.geofabrik.de/europe/czech-republic.html).

7. Last data set, that was necessary to acquire is data about air quality in Prague. This data is available on web page of [CHMU](http://portal.chmi.cz/files/portal/docs/uoco/web_generator/actual_hour_data_CZ.html).

### 2.3.2 Analyzing data

In order to use the data, first it was inevitable to understand the data.

1. The timetable data is stored in [GTFS](https://developers.google.com/transit/gtfs) format. That practically means that it consists of several text files, that are formatted as a table of values, first line being column names and all other lines holding the data. File named `stops.txt` holds information about all the stops in the system, including positions and names. Files `routes.txt` and `trips.txt` hold information about routes and trips operated in Prague and finally `stop_times.txt` holds information about departure and arrival times of individual trips. For every trip, there are several lines in file `stop_times.txt` describing every arrival and departure from stops visited during the trip. Every trip is repeating event, that can be happening on several days in the week and potentially for many weeks to come. However, the data is usually valid only for limited period of time, for a week or a little bit longer.

2. Delay data is stored in [JSON](https://www.json.org) format. This enables both human and machine readability. For each trip, that is currently on the way and is operated by other agencies that *Prague Public Transit Co. Inc. (DPP)*, there is one element about its position and delay. This element has information about last known coordinates given by latitude and longitude, last known departure delay time and departure delay times it had in previous stops.

3. Data about parking lots is also in [JSON](https://www.json.org) format. It consists of elements, each representing one parking lot. In the element there is name and id of parking lot as well as its location given by latitude and longitude and of course total number of parking places and number of currently empty ones.

4. The Czech Republic map data is not analyzed directly, but it is used as source data for Open Street Routing Machine [OSRM].
5. Air quality data holds tremendous amount of information about air quality in Czech Republic. It is possible to look up individual results of measurements for every meteoric station in the Czech Republic. There is a lot of different parameters including owner, classification of the station and levels of various particles like $SO_2$ or $NO_2$. Example of Prague’s data is shown in Figure 2.1.

### 2.3.3 Handling data format changes

All the data is relatively newly published and therefore there are a lot of changes made in the data format and the data itself. Unfortunately changes in the format can cause problems for the application trying to parse it. Some changes are just too significant to be predicted and correctly understood without prior knowledge of the change. This means program parsing the data must be regularly updated.

### 2.3.4 Parsing data

1. Reading data from text files is impractical for normal use and therefore we will create database everyday as new data is published. Database combines all the data files and changes the structure a little bit, so that it is more relevant to the task. Stops can be imagined as vertices in a graph and every trip forms several edges connecting the vertices. The data about trips and their arrivals and departures from stops is transformed into data about departures from stops. For every stop, there can be several departure directions, meaning from stop A there can be a bus going directly (without a middle stop) to stop B and another bus similarly to stop C. Departures for every stop and departure direction are sorted using departure times. Therefore, when we know that arrival to stop A is at time t, we can quite easily find departure, that has the smallest departure time, but is greater than t. And adding travel time we get arrival times to B and C via stop A.

2. Delay data must be parsed in real time, as the server is trying to find the best route. Only last known delay is used. Other information from the JSON file is ignored.

---

**Figure 2.1:** CHMU air quality data.

<table>
<thead>
<tr>
<th>Kraj.</th>
<th>Prague</th>
<th>Measurisation</th>
<th>Value</th>
<th>08.05.2020 13:00 - 20:00 UTC</th>
<th>$SO_2$</th>
<th>$NO_2$</th>
<th>CO</th>
<th>$PM_{10}$</th>
<th>$PM_{2.5}$</th>
<th>O$_3$</th>
<th>$O_3$ r. min.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALESE</td>
<td>Praha 1-Karlova</td>
<td>dopravní</td>
<td>0</td>
<td>30</td>
<td>72</td>
<td>4.1</td>
<td>29.3</td>
<td>109.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALESE</td>
<td>Praha 2-Legionska (hot spot)</td>
<td>dopravní</td>
<td>0</td>
<td>100</td>
<td>60.7</td>
<td>15.0</td>
<td>24.7</td>
<td>5.0</td>
<td>105.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALESE</td>
<td>Praha 4-Josef</td>
<td>mezinárodní</td>
<td>0</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>ALESE</td>
<td>Praha 2-Jiřího z Poděbrad</td>
<td>městský</td>
<td>0</td>
<td>30</td>
<td>72</td>
<td>3.1</td>
<td>3.0</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>ALESE</td>
<td>Praha 6-Kroměříž</td>
<td>dopravní</td>
<td>0</td>
<td>30</td>
<td>30</td>
<td>3.1</td>
<td>3.0</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>ALESE</td>
<td>Praha 3-Strahov</td>
<td>dopravní</td>
<td>0</td>
<td>30</td>
<td>30</td>
<td>3.1</td>
<td>3.0</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>ALESE</td>
<td>Praha 9-Vysočany</td>
<td>dopravní</td>
<td>0</td>
<td>30</td>
<td>30</td>
<td>3.1</td>
<td>3.0</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

---

22
3. A special stop is created in the database for each parking lot. Using OSRM, transfer times to the closest stops are calculated and the newly created stop is therefore connected with the rest of the public transport system. When searching connection from arbitrary position using car and then public transport, first car connection is found to one of the parking lots and then normal public transport search is performed from the special stop representing the parking lot in the public transport database.

4. The Czech Republic map data is not parsed in any way, but it is used as source data for OSRM.

5. Out of all the data about air quality only one summarizing parameter is extracted. It is a parameter describing overall air quality on scale from 1 to 6. Only data about Prague is used, in particular summary of Prague outskirts and Prague center.

2.4 Routing in public transport

When searching fastest route in public transport, we can imagine all stops and routes between them as a graph with vertices and edges. Every stop has an stop_id and a stop_name. But several stops can have identical stop_name, if they are part of one bigger station.

As input parameters we get one origin stop and one destination stop, given either directly by its stop_id or by its stop_name. In case of the stop_name there are more than one stop with this stop_name. That means, we search for fastest connection from any stop, that has the correct origin stop_name to any stop that has the correct destination stop_name.

If we want to search with middle stop set, e.g. A \rightarrow B \rightarrow C, then we can simplify it to searching first A \rightarrow B and then B \rightarrow C.

2.4.1 Routing algorithms

Now we have a graph representation, but what graph algorithm should be used?

It is important to realize, that from every stop there is usually just one direction, where we can go. This means that e.g. all buses leaving the station go in the same direction to the same next stop. All departures can be looked upon as oriented edges. Considering the previous observation, this means that for every vertex (stop) there is usually just one (or very few) oriented edge (departure direction) originating from this vertex.

That means, that number of edges $| E |$ is approximately $| V |$ or more accurately $| E | = c * | V |$, where $c$ is some small constant, that does not depend on $V$ or $E$. 
Floyd-Warshall algorithm can find shortest paths between every pair of vertices in a graph with positive or negative edge, but without negative cycles. All vertices are numbered from 1 to n, where n is number of vertices. In the first step it finds the shortest path between all vertices without using intermediate vertices. In second step only vertex with number '1' can be used as intermediate vertex. In next step vertices with number '1' and '2' can be used and so on. In the end all vertices are allowed as intermediate vertices and shortest paths are found for every pair of vertices.

Johnson’s algorithm is used to find shortest path between all pairs of vertices. Some edges can have negative weigh, but there can not be negative cycles. It uses Bellman-Ford’s algorithm to transform the graph into a graph with non-negative edges and then Dijkstra’s algorithm to find paths between all pairs of vertices.

Dijkstra’s algorithm is designed to find shortest path from one origin vertex to multiple target vertices. It works only with edges that has non-negative weight.

It always takes one vertex with the earliest arrival time, let’s call it S. The arrival time for S cannot be smaller, because any path to S via other vertices takes longer, because arrival times to other vertices are bigger. Then for every edge, leading out of S, we update the vertex’s arrival time on the other side of the edge. The new arrival time is S’s arrival time + weight of the edge. The update is performed only if the new arrival time would be smaller, than the precedent value. After going through all the vertices, we get the smallest arrival times for every vertex.

A* search algorithm is an extension of Dijkstra’s algorithm. It applies the use of heuristics in the search. In our case it enables us to use the great-circle distance in combination with minimal public transport speed as an estimate. Practically it means that Dijkstra will not search in all directions, but slightly more in the direction of the target stop.

Bellman-Ford algorithm is used for finding shortest path from single source to other vertices. In can handle negative weights of edges. However, it is much slower in comparison with Dijkstra. The algorithm relaxes all the edges in the graph in every step, which is the reason, why it can handle negative edges, but it also makes it a lot slower. Therefore, if we know, there are no negative edges, then Dijkstra is always better.

There are no edges with negative values in our graph, let alone negative circles. Therefore, advantages provided by Bellman-Ford, Floyd-Warshall or Johnson’s algorithm are not needed and in general cases the advantage is balanced by worse asymptotic complexity. Therefore, Dijkstra is more suitable for the given task. But we can also embrace a quite good heuristic using great-circle distance between current stop and the destination and use in A* algorithm to achieve further speed increase. That is why, A* was chosen as the most suitable algorithm for the implementation.

The definition of the algorithms and their advantages and disadvantages are available in the Cormen et al. [2009] and Mareš and Valla [2017].
2.4.2 Using delays

The access to delay data is one thing but using it properly can be hard. The problem is that delay of the bus is changing in time and if it is 20 minutes away, it can change significantly. And the change can be both ways either arriving even later or in rare cases arriving on time. In such case the connection might not work and catching following bus might be impossible. Or the user might miss the bus at the start if the delay was reduced.

On the other hand, there is no good way to predict these changes and without using the delay, we would have the same problem or even worse. It might be possible to use the delays in past weeks to predict possible delay time changes, but that is not goal of this thesis.

2.4.3 Handling transfers

There is unfortunately no good data about transfer times. Currently only transfer between stops with the same name, all part of one station, are allowed. Standard 120 seconds transfer time is used for all transfers.

2.5 Car and pedestrian routes

In the combined scenario, where individual car transport is combined with public transport, a car route must be found from the origin to some parking lot in Prague and then pedestrian transfer from this parking lot to some close public transport stop. There are two possible solutions. It is possible to use API of some service that can find the route or create our own solution.

2.5.1 Custom-created solution

Custom-created solution could offer some advantages as we could easily change the parameters of the search to our needs, especially when calculating transfer to public transport stops, that might be underground or poorly accessible. On the other hand, creating the solution would take a lot of effort and there are already some solutions solving the problem quite well.

2.5.2 Online providers

Another option is to use API of some online provider. This means to make a request to a provider’s server every time, we need car or pedestrian route. There is usually a small fee for every request, or it is free with limited number of requests per day/month. There are many different providers, but Google maps and Seznam maps are most relevant in the Czech Republic. Both provide quite accurate fastest route searching, but only limited number of requests can be made and if the number is exceeded a fee is charged for every request. In addition, we are not making normal route searching requests, but rather distance matrix searches, where one origin is supplied along with multiple destinations - parking lots coordinates. And because there could be possible even hundreds of different parking lots, there could also be hundreds of destination coordinates in the
distance matrix search. Problem is that these services do not allow unlimited number of destinations, e.g. Google allows up to 25 destination coordinates and Seznam does not support distance matrix searches at all. This means that for every search multiple requests have to be made and this further increases the price of the search.

2.5.3 Running own routing server

Last option is to download some free software that can solve the task and create our own routing server. This option is free and not limited with number of requests, but it consumes some server time and most importantly memory. However, it can save some time, because accessing API of some remote server can cause a delay and if our server is comparably fast, it will be faster to make a local search.

2.5.4 Comparison of providers

To compare the solutions three main categories were chosen. It is installation/implementation difficulty, speed of the solution and price.

- The most difficult solution is the custom-created one, because it would require nontrivial amount of time to implement, the second one is running own routing server, which can take some time to correctly configure and install. And the easiest is using API of online providers.

- When it comes to speed, all solutions are potentially comparable but accessing API over internet can be a big slow down. Therefore, best possibilities are custom-created server or running own routing server.

- Using online API free only for limited number of requests. If we plan to expand and have potentially thousands or even more of users and therefore requests per day, it can get quite expensive. On the other hand, running our own server, either custom-created or using some already created software, is much cheaper.

If we compare all the options, running our own routing server is the winner in almost all categories. Specifically, Open Street Routing Machine (OSRM), operating over Open street map data, seems like the best option. It is free and unlimited number of requests can be made. OSRM is used in many services and has broad user base.

2.6 Combining car and public transport routes

In order to find the fastest route between a given position and a public stop, we must find car transport arrival times to all available parking lots. It is not possible to find out which parking lot is the most suitable, because it depends on its relative location both to the origin position and the destination stop with regard to quality of public transport connection from this parking lot to the destination stop.

Fortunately, OSRM supports distance matrix requests, that means it can find distance between the origin and each parking lot at once. Due to the algorithms used, this does not take much longer than calculating distance to the farthest parking lot only.

When we get the arrival times to all parking lots, our final tasks is to find fastest route between one of them and the destination stop. Fortunately, A* can support multiple origins in the graph with different start times. This means, we can search from all the parking lots at once and when we reach the destination stop, we can just backtrack the route to find the parking lot, which was used to get to the destination the fastest.

It is also necessary to connect parking lot position with some of the neighboring stops. Otherwise we would not be able to find connection between parking lot and the any stop, because there would be no edges connecting the two graphs. Again OSRM is used to find walking distance between parking lots and few of the closest stops.

2.7 Different OS on client devices

There can be different operating systems on the client devices. Most common operating systems on mobile device are currently Android (75.3%), iOS (22.4%), KaiOS (0.8%) and many others having less than 0.5% market share. Because the hard task is handled on the server side, client consists of only graphical interface and network communication with server. Client implementation was created only for Android devices as they have the biggest market share. But it is very easy to implement client application for other operating systems, if necessary.
3. User documentation

This section is about how to correctly use the client application and how to correctly use the API provided by the server.

3.1 Installing via APK

Mobile client support only Android device with API level higher or equal to 19. It will most probably not be able to install the application on devices with lower API level or if successful, the application might crash or it might behave unexpectedly.

Installation via APK is very simple. Using Google’s Play Store[^1] it is just a matter of clicking install button. If the APK was downloaded without using Play Store, than it is necessary to allow usage of third party APK in the mobile device. Usually it can be found in setting under security.

APK stands for Android application package. It is a format used for distribution and installation of Android mobile apps. APK is a type of archive file based on zip format.

3.2 Initial start up

When application is installed and started for the first time user is introduced to the graphical user interface.

3.2.1 Graphical user interface

Graphical user interface consists of two fields, one for origin stop and one for destination stop, time field, date field, ”Search” button and ”Download hints” button and finally a switch, that hides origin stop button. All is clearly visible in 3.1.

3.3 Basic usage

Usage is quite easy, all that has to be done is to fill the origin and destination stop name and eventually change time and date and click ”Search” button and wait for the result. In this case normal search is made and only public transport will be used in the route.

If you want the application to automatically complete stop names, you have to click ”Download hints” button. It will download all names of stops in the system and it will get much easier to type the correct names.


28
Figure 3.1: Graphical user interface.
Figure 3.2: Combined interface
3.4 Advanced usage

It is also possible to find route using individual car transport first and then continue using public transport. In this case current location is the origin position. This mode is entered by switch the "Combined" switch. You can see in 3.2 that only one field for stop name is left, because origin stop name is no longer needed.

3.5 Results

After clicking the "Search" button, the request is sent to the server. When the server finds corresponding connection the application will show the results as can be seen in 3.3 or 3.4 for combined search. It is quite simple. Every connection consists of several elements. Each element is displayed and consists of origin and destination location and time and used mean of transport. After clicking the back button, you can return back and make another search.
Figure 3.3: Normal result
Figure 3.4: Combined result
3.6 Limitations

Searches are limited to one week period, as the public transport data is usually valid only for one week. If departure time is more than a week in the future no connection will be found. On the other hand, if departure time is back in the past, some connection will be found, but it will depart in the present time. More precisely it will use the earliest departure present in the data.

Currently only one connection is found for each request and it is only possible to search connection by departure time (setting arrival time is not possible).

Location must be known to the device, when performing combined search. If location is yet unknown, the search can not be executed. Warning message is displayed in such case.

3.7 Typical use cases in GUI

In this section, it is explained how main use cases are introduced in mobile application.

3.7.1 Public transport search

First use case is public transport search. That means searching connection from one public transport stop to another public transport stop. In order to make this kind of search, combined/using car switch must be turned off.

3.7.2 Combined search

Seconds use case is combined search. In this case it is presumed, you start the transport using a car. Therefore connection to some appropriate park lot near or in Prague is found and than the connection is continued via public transport in Prague. The current location of the mobile device is used to determine starting position and also public transport stop name is required as destination. In order to make this kind of search, combined/using car switch must be turned on.

3.8 Server API

Apart from using supplied client application, it is also possible to use directly the API of the server. It can be used to create another client application for different mobile operating system or for any other device with internet connection. It is also fully possible to create a web page using the API and accepting request from users connected via browser.

The server is running under domain name kbely.mattty.cz and is accepting requests on port 10548.
3.8.1 Request and result format

Both request and result is formatted using Google’s Protocol buffers. File message.proto is an attachment of this thesis and describes the used Protocol buffers format.

Basically stop names or stop name and location, time and information, if delays should be used, is supplied. It is also possible to set a list of middle stops.

Result is also defined in message.proto and consists of elements describing the route, that means used routes, transfers and times of transfers. Other than that, elapsed time in milliseconds is returned as well. If something went wrong, than boolean parameter error is set to true and string errorMessage describes the details.
4. Programmer documentation

This section is intended for any programmer that will be trying to understand and extend this project.

4.1 Building Qt application

Qt library source files consist of C++ headers and source file and also one special file with .pro suffix. This file is used to correctly build the project. Using tool called *qmake*, we can create a *Makefile* for the project and then build it using *make*.

4.2 Database creator

Main goal of Database creator is to process all data files containing information about public transport timetables, transfer times and park lots. After that a [SQL](https://en.wikipedia.org/wiki/SQL) database is created to hold all the information in different format, that is more suitable for performing route searches.

**SQL** (Structured Query Language) is programming language designed to manage structured data, where there are relations between different entities of the data. That is exactly what we need as it corresponds well with data structure of public transport timetables.

4.2.1 System requirements

Application is written in C++ 11 using Qt library. It is tested under Linux/Gentoo. Versions used were 8.3.0 for GCC 2.31 for binutils and 2.2 for glibc. The program needs minimum of 4 GB of operating memory.

4.2.2 Installing application

Shell script *update_database_creator* will build the application. It uses the source files to build the application. After it is successfully done, another script *create_database* downloads timetable data and then start the actual Database creator.

4.2.3 Shell scripts

The Database creator also downloads additional files, in particular data about park lots and it also makes request to [OSRM](https://openstreetmap.org) to get transfer times between park lots and public transport stops. For this purpose, following scripts are used *create_token*, *get_park_lots* and *get_osrm_response*. *create_token* gets token from the server providing [API](https://en.wikipedia.org/wiki/Applicationプログラミング) for park lots data. *get_park_lots* than uses this token to get the actual data. Script *get_osrm_response* makes a request to [OSRM](https://openstreetmap.org) server. For this purpose, file *osrm_request*, which is created by Database creator itself,
is used. There are also some scripts written directly in the C++ code, which is possible quite easily thanks to Qt library.

### 4.2.4 Processing data

1. Data about timetables is parsed from GTFS format into C++ structures.

2. In addition data from special file `search_transfer.txt` is also parsed into C++ structures. The file `search_transfer.txt` holds information about transfer times and is used as update to standard 120 seconds transfer times.

3. Information about park lots locations is downloaded and special stop is created for each park lot position. These stops are added to the list containing data about ordinary public transport stops.

4. In order to join special park lot stops with the rest of the stops, we need some sort of transport between them. Only possible way to get from park lot to normal stops is to walk there and therefore we create transfers between park lots and public transport stops. For each park lot several transfers are created to the nearest stops.

5. After all the necessary data is parsed, we created list of departures for every stop and every direction, where we can go from this stop. Departure times and `trip_id` are stored in file `stop_times.txt`. Corresponding dates are defined by `trip_id` and corresponding `service_id`. Departure times can be higher than 24 hours. That means departure times like 26:01:05 exist. This is why, date defined by `trip_id` can be different than actual departure date. Reasoning behind this is, that trips, that set out after midnight, are still logically bonded to the previous day and hence the 26 hours departure times are used. We need to get them to format, where we have departure for specific stop and directions sorted by departure time.

6. Next step is to save the data into SQL database. This means information about stop, routes, trips, transfers and departures. Only information currently needed to search connection is saved into the database, that means not all data read from GTFS files will necessarily end up in the database. For departures two tables are created. The one called `departureDirections` holds information about all departure directions, that means pair of `stopId` and `targetStopId` and `departureDirectionId`. The other table is called `departures` and holds information about departures for every `departureDirectionId`, that means `departureTime`, `travelTime` and `tripNumber`.

7. In the end Database creator also creates a text file containing names of all the stops. This list is than available for download in client application and is used to automatically complete stop names for the user.

### 4.3 Open Source Routing Machine

This external and open-source project. And not part of my work.
OSRM is a route planner designed to be used with data from OpenStreetMap project. OSRM is written in C++ and is available on platforms like Linux, macOS, and FreeBSD.

This open-source application is used to find routes between arbitrary locations, defined by coordinates. Every instance can use only one type of transport. We need to search for both pedestrian and car routes. Pedestrian routes are currently needed only, when creating SQL database, because we need to determine transfer times between park lots and public transport stop. On the other hand, car routes are only needed when the server application is search combined route using individual car transport.

In our case only searches in the Czech Republic are supported and therefore only Czech Republic OpenStreetMap data is used. Both car and pedestrian instance needs less than 1 GB of operating memory, when running.

4.4 Public Transport Connection library

Public Transport Connection library (PTC) is part of this project but can be used independently as well. It is used to find fastest connection between any two stops in Prague. SQL database created by Database creator is used as data source. Apart from the database, data about delay times of some of the public transport routes is used as well.

4.4.1 System requirements

Library is written in C++11 using Qt library. It is tested under Linux/Gentoo. Versions used were 8.3.0 for GCC, 2.31 for binutils and 2.2 for glibc. The program needs minimum of 1 GB of operating memory.

4.4.2 Basic ideas and solutions

1. PTC is using SQL database as a source. Using Qt library, we can easily access data in the database. Data about departures and stops is read into memory during initialization, because C++ containers are more efficient, when searching for suitable departure from a stop. This happens in class ConnectionSearch in function initialize.

2. We use A* algorithm, which is an extension of Dijkstra’s algorithm. The difference is, that A* uses heuristics to estimate arrival time in the destination. In our case this is done in function calculateEstimatedGoalTime_ of class ConnectionSearch. We use coordinates to calculate great-circle distance between the stop and the destination. After that we divide the calculated distance by estimated speed. Important thing is, that the estimate arrival time must always be lower than reality. This means estimate speed must be higher than the real one. But on the other hand, closer to reality it gets, the faster the algorithm is. The estimate speed was acquired experimentally.
3. For every stop we create object represented by class `GraphStop`, that owns all the import information about the stop, needed in the algorithm. These stops are stored in QHash named `graphStops` and indexed by `stop_id`. Another container of type GraphQMap and named `graphHeapStops` holds pointers to every stop and is indexed by arrival times, it is used in A* to find a stop with the lowest arrival time. GraphStop objects are altered during the search and they have to be reset, before making another one.

4. In every step of the algorithm defined in `ConnectionSearch` class, we take stop with lowest arrival time and relax all edges coming out of it. There are always two types of edges. One is transfers to stops, that are usually all part of one station or transfer to special parking lot stops. The other is one stop connections to neighbor stops connected via buses, trams, metro etc.

If the resulting arrival time is lower than the best previously found, we update the stops arrival time and arrival route type. For the purpose of neighboring stops, we try to find the best suitable departure. However, when we use delays, we have to look at several departures, because they order can change due to different delay times.

In the end of the step, we close the stop and find next one with lowest arrival time.

5. We continue as long as there are other opened stops. When there is no more opened, it means, that there is no connection to be found and error message is returned by function `startSearching`. Otherwise we should sooner or later find one of the destinations stop. In that case we return the `stop_id` of the stop.

6. When we update arrival in any stop, we set arrival time, arrival route, predecessor and other important values. This allows us to reconstruct the route in function `createPath`, when we have reached the destination stop. We simply go backwards and add all the stops along the way in a QVector.

This means, we can easily join two compatible paths together and parse the path to elements. Struct `Element` is a user-friendly format to display the connection. Basically, it means, we do not display all stop names and arrival times along the way, but only those where we change the transport type.

### 4.5 Server application

Server application uses [PTC](#) to search fastest connection between two stops or location and stop. Apart from the [PTC](#) current information about [P+R](#) park lots is also used, as well as data about air quality.
4.5.1 System requirements

Application is written in C++ 11 using Qt library. It is tested under Linux/Gentoo. Versions used were 8.3.0 for GCC, 2.31 for binutils and 2.2 for glibc. The program needs minimum of 1 GB of operating memory, most of it because it uses PTC.

4.5.2 Installing application

Shell script update server will build the application. It uses the source files to build the application. After it is successfully done, another script server will download current delay data and start the actual server application.

4.5.3 Basic ideas and solutions

1. Server application is using PTC library, which is fully responsible, as its name suggests, for finding the connection in public transport. It is started and initialized immediately after server start up.

2. Server is listening on a port 10548 on address kbely.mattty.cz. Server can handle only one request at a time and has queue of 128 requests. If more requests are made, they will receive cannot connect error. Class ConnectionServer is responsible for handling internet communications.

3. We are using Protocol buffers to format data sent over network. That means that incoming requests have to be parsed first. And of course, result sent back must be converted back into Protocol buffers structures. Protocol buffers format, that was used, is described in file message.proto. It is used to generate C++ header and source file, that handles the formatting.

4. For every request it is determined, if it is a normal or combined request. If it is a normal request, which means only Public Transport is used, then we pass the parameters to the PTC. In case of combined request, we call function combinedSearch of class Search, that is responsible for solving combined requests.

5. Function combinedSearch calculates arrival times to all available parking lots. Every parking lot is a special stop in public transport data. Therefore, these arrivals can be used as origin stops in PTC. After connection from one parking lots to the destination is found, we receive a PublicTransportConnection::Result object. This object represents the public transport route. We join it together with the individual car transport route to get the whole route using addDriveToParkLot function of class Search.

6. To be able to find car transport to all parking lots, we must first download data about P+R and then use OSRM server to find out how long it takes to get there from original position.

7. Data about air quality is also downloaded to calculate penalty for individual car transport. In case of poor air quality, public transport is preferred.
4.6 Client application

This subsection will sum up implementation details of the client application.

4.6.1 Building application

The application code is written in Java and developed in Android Studio. Building the application is quite easy. The application uses Google’s protocol buffers to easily and safely communicate with server application. Android Studio supports protocol buffers and using it is as easy as adding few lines in files build.gradle, one located directly in the root directory and other in subdirectory app of the project.

When testing the application, we can connect an Android mobile device to the computer and run the application in the debugging mode directly on the mobile device. It is as easy as clicking run in the Android Studio and choosing the device correct device from the list. Other option is to use an emulator, that is part of the Android Studio. It can be little bit slower and it will take up quite a bit of the computer’s memory. But it enables to test the application on several Android API level versions quite easily.

When application is ready, it is necessary to create a signed APK to deploy it on user’s mobile devices. This can be done also through Android Studio. It is again very easy and practically all that is necessary is to choose a key by which the APK is signed. After that application can be deployed to Google’s Play Store and users can easily download it.

4.6.2 Basic ideas and solutions

There are two kinds of search modes. One is normal public transport search. And the other is combined search using car travel and public transport. There are two classes corresponding to these modes. The first one is PublicTransportSearchParams and the other one is CombinedSearchParams. The difference is that in the first case two public transport stop names are send to the server in the seconds case only one stop name is sent and instead of the origin stop name coordinates in the form of latitude and longitude are sent.

Server is located at the domain name kbely.mattty.cz. Data is sent using socket connected to port 10548. Protocol buffers ensures that the data will be correctly interpreted on the server side and also the result sent back from the server. All this is handled in class ConnectionClient.

The result data structure is parsed to instance of Java class Result. This structure consists of list of elements. Each element describes one part of the route, where in each part only one specific type of transport is used.

New activity is created in order to display the result and each element is displayed as one line describing one part of the route. This is implemented in class SearchResultActivity.
Button download hints enable the user to download special text file with names of all stops in Prague. This file is than used to display hints, when typing the name of origin or destination stop. Most of this feature is handled by class DownloadFile.
5. Testing

Both client and server application has been tested in many different ways. Client application was tested for usability, Android API levels compatibility and internet usage. Server application was tested for speed.

5.1 Client Usability

In order to test usability of the client application, a Google Form was created with questions about GUI of mobile application. The results are shown in the form of graphs.

Results

Questions: 5.1 5.2 5.3 5.4 5.5 5.6 5.7 5.8 5.9 5.10
How do you like the look of the app?

8 responses

Figure 5.1: Question number 1

How do you appreciate the speed of the application?

8 responses

Figure 5.2: Question number 2
What features did you use?

8 responses

![Bar chart showing feature usage.]

Figure 5.3: Question number 3

How do you appreciate data and storage usage efficiency of the application?

8 responses

![Pie chart showing data storage efficiency.]

Figure 5.4: Question number 4
How do you like the app design?
8 responses

![Pie chart showing responses to How do you like the app design?](chart5.5.png)

Figure 5.5: Question number 5

Does the application control suit you?
8 responses

![Pie chart showing responses to Does the application control suit you?](chart5.6.png)

Figure 5.6: Question number 6
Can you describe a situation in which our app is most useful?

2 responses

<table>
<thead>
<tr>
<th>Good use when changing travel plans</th>
<th>25%</th>
</tr>
</thead>
<tbody>
<tr>
<td>When I travel somewhere I don't know</td>
<td>75%</td>
</tr>
</tbody>
</table>

Figure 5.7: Question number 7

Does the application offer new functionality compared to other apps?

8 responses

Figure 5.8: Question number 8

Would you recommend this app to your friends?

8 responses

Figure 5.9: Question number 9
5.2 Different Android devices and API levels

The client application has been tested on several Android devices with different API levels from 19 to 28. In all cases hints were downloaded and both normal and combined search was performed. No problems have been encountered.

### Results

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Model</th>
<th>Android API</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nokia</td>
<td>3.1</td>
<td>level 28</td>
<td>OK</td>
</tr>
<tr>
<td>Huawei</td>
<td>Honor 3C</td>
<td>level 25</td>
<td>OK</td>
</tr>
<tr>
<td>Huawei</td>
<td>Honor Holly</td>
<td>level 19</td>
<td>OK</td>
</tr>
<tr>
<td>Xiaomi</td>
<td>Redmi 4x</td>
<td>level 25</td>
<td>OK</td>
</tr>
<tr>
<td>Xiaomi</td>
<td>Redmi 3</td>
<td>level 22</td>
<td>OK</td>
</tr>
<tr>
<td>Xiaomi</td>
<td>Mi3</td>
<td>level 23</td>
<td>OK</td>
</tr>
<tr>
<td>Samsung</td>
<td>Galaxy J3 (2016)</td>
<td>level 22</td>
<td>OK</td>
</tr>
</tbody>
</table>

5.3 Multiple requirements

Making up to 100 of parallel requests and test if server can handle it and how fast it works under heavy load.

### Results

Server has a queue, that can keep at most 128 requests waiting. When 100 parallel requests are made, all of them end up in the queue and are processed one by one. If more than 128 requests are made, then all new requests are denied.
When testing this functionality, everything was according to expectations and 100 parallel requests were handled without problems.

Speed of the server was also measured. This was notebook with following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>4x Intel(R) Core(TM) i7-3517U CPU @ 1.90GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>7.6GiB of RAM</td>
</tr>
<tr>
<td>OS</td>
<td>Linux Gentoo</td>
</tr>
<tr>
<td>GCC</td>
<td>version 8.3.0</td>
</tr>
<tr>
<td>binutils</td>
<td>version 2.31</td>
</tr>
<tr>
<td>glibc</td>
<td>version 2.2</td>
</tr>
</tbody>
</table>

Client and server applications were both running on this device. Therefore, there is no interference caused by internet communication. Several test cases were created as you can see in the following table and 1000 request times were recorded for each of them.

<table>
<thead>
<tr>
<th>Test</th>
<th>Mode</th>
<th>Delay</th>
<th>Origin</th>
<th>Destination</th>
<th>Middle stops</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA</td>
<td>Normal</td>
<td>No</td>
<td>Důst. domy</td>
<td>Malos. nám.</td>
<td>Florenc - C</td>
</tr>
<tr>
<td>NB</td>
<td>Normal</td>
<td>No</td>
<td>Důst. domy</td>
<td>Malos. nám.</td>
<td>Křižíkova</td>
</tr>
<tr>
<td>NC</td>
<td>Normal</td>
<td>No</td>
<td>Háje</td>
<td>Vysočany</td>
<td>None</td>
</tr>
<tr>
<td>ND</td>
<td>Normal</td>
<td>No</td>
<td>Důst. domy</td>
<td>Malos. nám.</td>
<td>None</td>
</tr>
<tr>
<td>NE</td>
<td>Normal</td>
<td>No</td>
<td>Čakovice</td>
<td>Malá Chuchle</td>
<td>None</td>
</tr>
<tr>
<td>NF</td>
<td>Normal</td>
<td>No</td>
<td>Prosek</td>
<td>Ládví</td>
<td>None</td>
</tr>
<tr>
<td>DNA</td>
<td>Normal</td>
<td>Yes</td>
<td>Důst. domy</td>
<td>Malos. nám.</td>
<td>Křižíkova</td>
</tr>
<tr>
<td>DNB</td>
<td>Normal</td>
<td>Yes</td>
<td>Důst. domy</td>
<td>Malos. nám.</td>
<td>None</td>
</tr>
<tr>
<td>DNC</td>
<td>Normal</td>
<td>Yes</td>
<td>Háje</td>
<td>Vysočany</td>
<td>None</td>
</tr>
<tr>
<td>DND</td>
<td>Normal</td>
<td>Yes</td>
<td>Důst. domy</td>
<td>Malos. nám.</td>
<td>None</td>
</tr>
<tr>
<td>CA</td>
<td>Combined</td>
<td>No</td>
<td>50.1360, 14.0995</td>
<td>Palmovka</td>
<td>None</td>
</tr>
<tr>
<td>CB</td>
<td>Combined</td>
<td>No</td>
<td>49.9703, 14.5153</td>
<td>Malos. nám.</td>
<td>None</td>
</tr>
<tr>
<td>CC</td>
<td>Combined</td>
<td>No</td>
<td>49.4146, 15.5916</td>
<td>Petřín</td>
<td>None</td>
</tr>
<tr>
<td>DCA</td>
<td>Combined</td>
<td>Yes</td>
<td>50.1360, 14.0995</td>
<td>Palmovka</td>
<td>None</td>
</tr>
<tr>
<td>DCB</td>
<td>Combined</td>
<td>Yes</td>
<td>49.9703, 14.5153</td>
<td>Malos. nám.</td>
<td>None</td>
</tr>
<tr>
<td>DCC</td>
<td>Combined</td>
<td>Yes</td>
<td>49.4146, 15.5916</td>
<td>Petřín</td>
<td>None</td>
</tr>
</tbody>
</table>

Results are displayed in box plot graph showing summary of all measured values. Following are two graphs. 5.11 is displaying values for search in normal mode without delay. 5.12 is displaying values for all the other tests.

### 5.4 Client internet usage

Because mobile data is quite expensive in the Czech Republic, it is important to minimalize their usage. It of course depends on the type of the request and how long the final route is. The longer the result, the more data is used. But as you can see in the results section, in all cases data consumption is minimal and the application uses almost no data at all. Part of the results is also comparison to other applications able of comparable tasks.
Results

Measured results are only indicative, because real data consumption depends on many factors. Results for our solution can be seen in Figure 5.13.

Measuring data consumption of other application is not so easy and not so precise. Therefore, only indicative measurements were performed. In both cases start up and one request was made and then the (identical) request was repeated. The request was for a connection between "Důstojnické domy" and "Malostranské náměstí". Possible reasons for much bigger data consumption are download of metadata and/or advertisement.

**IDOS**: Start up and request used 158,000 bytes and repeated request used another 20,000 bytes.

**Pubtran**: Start up and request used 170,000 bytes and repeated request used another 60,000 bytes.

Figure 5.11: Normal mode speed results.
Figure 5.12: Other speed results.
Figure 5.13: Data consumption.
Conclusion

First, we determined what data is available and what innovational project could be created with it. Though there are a lot of solutions for finding routes in public transport and even in combination with walking, there is currently no solution for finding route in public transport in combination with individual car travel and parking. It is even more surprising, when we consider hundreds of thousands of cars and their drivers coming to Prague every day for work, school or other reasons.

We assessed the user requirements for the application and analyzed the open data available as well as some related project and application in both Prague and the rest of the World.

Application design was created. We summarized potentially useful algorithms and chose the most appropriate. We decided, what module scheme to use and what programming language should be used to implement the algorithms. We understood and parsed open data and created the implementation that uses it. We also used external software to handle partial problems.

We documented the source codes and described implementation problems and solutions. We also described the application [GUI] and how to correctly use it and even how to use [API] of the server itself for someone, who wants to create client application for other systems.

Finally, the whole solution was tested in several different ways. First the client application was tested regarding its usability, [API] levels compatibility and internet communication efficiency, considering mainly mobile data usage. Than the server side was tested by creating multiple [API] requests at the same time and measuring time of the computing.

There are many possible extensions. The client application could be expanded to offer more preferences regarding the search parameters and even other client applications could be created for other operating systems. The server side might be able to search not only above public transport and car travel data, but also use cycling, walking, car-sharing, bike-sharing or any other means of transport possible in Prague. Currently the city of Prague is planning to create complex intermodal planner using all of the above mentioned. This work might become a starting point for the implementation.
Bibliography


List of Figures

1.1 Dpp.cz results. .................................................. 12
1.2 Fuzee results. .................................................. 13
1.3 IDOS results. .................................................. 14
1.4 Pubtran results. ................................................. 15
1.5 Mapy.cz result. ................................................ 16
1.6 Helsinki Region Transport result. ......................... 17

2.1 CHMU air quality data. ....................................... 22

3.1 Graphical user interface. ..................................... 29
3.2 Combined interface ........................................... 30
3.3 Normal result .................................................. 32
3.4 Combined result ................................................ 33

5.1 Question number 1 ............................................. 44
5.2 Question number 2 ............................................. 44
5.3 Question number 3 ............................................. 45
5.4 Question number 4 ............................................. 45
5.5 Question number 5 ............................................. 46
5.6 Question number 6 ............................................. 46
5.7 Question number 7 ............................................. 47
5.8 Question number 8 ............................................. 47
5.9 Question number 9 ............................................. 47
5.10 Question number 10 .......................................... 48
5.11 Normal mode speed results. ............................... 49
5.12 Other speed results. ........................................ 50
5.13 Data consumption. .......................................... 51
Glossary

API  Application programming interface. iii iv 2 3 20 21 28 36 41 43 48 53

APK  Android application package. 1 28 41

CHAPS  CHAPS is company concentration on software solutions for personal transport. 11 14

CHMU  Czech Hydrometeorological Institute. 21 22 55

DPP  Prague Public Transit Co. Inc.. 10

GCC  GNU Compiler Collection. 39 38 40

GPS  Global Positioning System. 10

GTFS  General Transit Feed Specification. 20 21

GUI  Graphical user interface. 2 4 34 43 53

IDE  Integrated development environment. 20

JSON  JavaScript Object Notation. 21 22

OS  Operating system. 1 27

OSRM  Open Source Routing Machine. 21 23 26 27 36 38 40

P+R  Park and Ride parking lot. 20 39 40

PID  Prague’s integrated transport. 20

PTC  Public Transport Connection library. 38 39 40

ROPID  Regional organizer of Prague’s integrated transport. 3 19 20

SQL  Structured Query Language. 4 19 36 37 38
A. Attachments

A.1 First Attachment

Attachment of this thesis is a CD with all source codes described in this project as well as generated documentation and electronic version of this thesis.