FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

František Mrkus

Processing of time tables

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis:	RNDr. Jiří Fink, Ph.D.
Study programme:	Computer Science
Study branch:	Programming and software development

Prague 2024

I declare that I carried out this bachelor thesis on my own, 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.

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 .... date ....

Author's signature

To my family, my supervisor, and public transport fans. Thanks for everyone who I could ask for help and motivation.

Title: Processing of time tables

Author: František Mrkus

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jiří Fink, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: A goal of this thesis is to create an open-source application which could serve as foundation for public bus transport analysis and organizing, while directly operating with timetables in a JDF format for a comfortable workflow. The application is centered aroud bus scheduling for public transport organizers and agencies,including related functions such as displaying timetable sheets and departure/arrival lists, map visualization of the planned routes, and creation of custom timetables. All of these features were sucesfully implemented and tested on real-world data.

Keywords: bus scheduling, public transport, optimization, spreadsheet timetable

Název práce: Zpracování jízdních řádů

Autor: František Mrkus

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Jiří Fink, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Cílem této práce je vytvořit open-source aplikaci, která by mohla sloužit jako základ pro analýzu a organizaci veřejné autobusové dopravy, přičemž přímo pracuje s jízdními řády ve formátu JDF, pro zajištění pohodlného pracovního postupu. Aplikace je zaměřena na plánování autobusových spojů pro organizátory a dopravce veřejné dopravy, včetně souvisejících funkcí, jako je zobrazení jízdních řádů a seznamů odjezdů/příjezdů, vizualizace plánovaných tras na mapě a vytváření vlastních jízdních řádů. Všechny tyto funkce byly úspěšně implementovány a otestovány na reálných datech.

Klíčová slova: rozvrhování autobusů, veřejná doprava, optimalizace, vývěsný jízdní řád

Contents

Chapter 1

Introduction

1.1 Motivation

We intend to handle the bus scheduling problem, starting in the Czech Republic. The applications used for scheduling usually are either commercial or require simpler format than timetables, which are provided in the form of open data.

An initial scheduling problem was provided to us by a transport agency for a relatively large region - Pilsen, Czech Republic, which we settled on as a limit for the data our application should be able to handle. The data were provided in the JDF 1.11 format [1], so naturally a conversion tool was needed.

Since the problem turned out to be more complex than expected, we decided to integrate the helper functionalities into one application - Bus timetable processor (TTP).

We also needed to provide a way to create new timetables, as while editing the JDF files directly is possible, it still appeared to be user-unfriendly.

We decided to delegate this task to publicly available spreadsheet software, supporting the import of spreadsheet files into TTP, which would also handle the scheduling in this format by converting it to JDF.

Ultimately, the bus scheduling was kept only in its simplest form, and the focus was made on making the software more accessible to the public, as while the scheduling is well-researched, the tools for working with JDF data are less common.

As a potential end goal, the software is expected to be usable by transport organizers or agencies as a proposal of a new timetable, or to optimize the scheduling of the buses in the existing timetable.

In the following sections, we elaborate on the bus transport system and the

[1] The “F“ stands for Format, so forgive us for the RAS syndrome in this thesis. The format is more explained in 1.4

software tools used in this field.

1.2 Bus transport

A bus transport system is the most widespread (by area) form of subsidized public transport in Europe.

Usually the system of regional transport is ordered by the regional government (organizer, in further sections) who signs a contract with a transport agency. The amount of money paid to the transport agency is usually fixed - based on the amount of kilometers driven, while the distribution of fare revenue (based on the amount of passengers) transported can be received either by the organizer or the agency ([1]).

The former offers more economic stability to the transport system, the latter motivates the agency to provide better service on its own.

A shorthand distinction of these modes is calling them "brutto" and "netto" contracts, respectively [2].

An agency serving urban transport can be either contracted by the city using the same system as regional transport, or the city can also own the transport agency, which can offer better control of the transport service.

Regardless of the contract type, there will be always an interest to minimize the operational costs.

In this thesis, we focus mostly on software capable of handling regional transport in the Czech Republic, and present a software tool, further named as TTP (timetable processor), which can be used to process the timetables of the buses in openly available data format JDF 1.11 and provide relevant functions for the organizer.

1.2.1 Trip planning

The most common routes for bus transport are usually based on the directions of the car travelling. However, while cars can use highways to travel to the desired distance in the shortest time, regional transport is based on efficiency of how many passengers it can transport, therefore the routes are usually aimed through cities and use more stops. An exact mathematical models on network planning are analyzed for example in [3].

Its goal is to allow most people to get onto bus stop in reasonable distance from their home and get off at reasonable distance from their destination. This destination, in the case of regional transport, is usually a bigger city which contains many workplaces, schools and other facilities. The route planning is therefore based on providing reasonably short route while also visiting as many cities as possible. As some of the facilities have fixed working hours, the transport system is first designed to allow people to get to these destination at given time.

As a second priority, the organizer should provide transportation to higher modes of transport, such as train stations or airports. Again, usually the bus transport must adhere to fixed schedules of these.

Finally, many trips are not based on any fixed time; they are spread around the day in such a way to avoid long gaps, allowing people to use public transport for their daily activities.

How often the bus actually come to given location is usually based on the amount of potential passengers, as more passengers also means more revenue for the organizer. Usually the system is not profitable, but some trips can actually be profitable, which can be used to subsidy the lower populated areas.

In general, we assume the system as a whole is aperiodic during the whole day, but the trips usually repeat every workday (most systems have two main schedules - one for workdays and one for weekends).

The easiest way of cutting costs, which we also discuss in our work, is to use as least buses as possible, which means that the buses must be used for as many trips as possible.

Furthermore, it is also necessary to minimize the distance and time without passengers (deadhead trips). Notably this is usually what also the organizers are already taking into account. For example, if a bus from a stop A to a stop B ends its trip at 10:50 [2] and stop B is not served by many trips during this time, then a trip from B to A should not be scheduled to start earlier. This can be generally called "integration of timetable planning and bus scheduling", mathematically analyzed in publications such as [4].

However, the proposed scheduling might not always be feasible or optimal, and generally the agency is responsible for both adherence to timetable and following the local laws regarding bus transport (not the organizer).

In our TTP, we do not suggest any change to the timetables, and focus only on optimal bus scheduling with timetables which cannot easily be changed.

1.2.2 Bus scheduling

The bus scheduling is a complex task, as it must take into account many factors. In the thesis and TTP, we only consider the simple tasks of scheduling buses themselves (without regards to the regulations of the drivers). The buses also are considered to be of single type, meaning all buses have uniform operational costs and can be assigned to any trip.

[2] we use 24-hour format

We generally consider two cases. Both of these have in common to minimize the time spent on deadheads and the amount of buses. What differ is how the buses get to their respective first and last trips (called pull-out and pull-in, respectively.). In the first case, we assume that the buses are already in the depot at the beginning of the day and return to the depot at the end of the day. An agency would have only one depot, which would be the starting and ending point for all buses. This leads to single depot bus scheduling problem (SDBS[3] for short), which can be calculated in polynomial time [5].

Having multiple depots - creating multiple depot bus scheduling problem (MDBS) - increases the difficulty significantly - proven to be NP hard by a reduction to multicommodity flow problem [5]. An extreme case of MDBS might be that we do not consider depots at all, where we must assume the buses are supposed to return on their starting space.

This is a practical problem which was stated as an actual requirement of a transport agency during the initial research (personal communication). The buses can be used for multiple days without returning to the depot, while the timetables are also scheduled in a way to make first trips of a day from suburban areas to bigger cities and last trip from bigger cities to suburban areas.

Removing the depot requirement can be seen as a way to save on pull-out and pull-in costs, as the buses would be nearby the place of their first and last trips. However, this appears to simply be an extreme version of MDBS, as we need to consider every initial stop (of all trips) as potential depot with infinite capacity. The problem stated this way was not found in literature, so we have decided to solve such problem in a heuristic way which scales well with the amount of possible terminal stops (starting and ending) instead of solutions which would get more complex with the amount of depots, such as in [6].

1.3 Bus timetable software

The transport agencies usually use their own software to create the actual timetables [4], where the machine-processable format is usually not shared with the public. Such timetables are published as human-readable PDF, in current Czech legislature they have to be on all stops the timetable relates to (=where at least one stop event from given timetable takes place, except for get off only stops [7] § 18,1f 111/1994 Sb.)

These timetable data in machine processable format are obviously necessary for scheduling of their buses and drivers, as well as for the internal bus systems

[3] More generally SDVS, with V standing generally for vehicle. Do not confuse with vehicle routing problem.

[4] A commercial example: https://www.tvorbajizdnichradu.cz/

(selling tickets, announcing stops).

1.4 Data formats for timetables

As of 2023, the most common format for timetables is GTFS[8]. It is an open source format developed by Google, which is used by Google Maps to show public transport connections.

In the Czech Republic, another format has been standardized, namely JDF (Jednotný datový formát, in English: Unified data format). The transport agencies are obliged to provide their timetables to centralized agency (CHAPS), which allows the public to view all the timetables on web [5] as well as allow downloading of the raw data in the JDF format [6]. As for 2023, the specification of the JDF is in format 1.11.

1.5 Software using public data

There are many software solutions which use the public data to provide the passengers with information about public transport. The most useful are the connection planners, such as the one available on Google Maps[7] which allow the user to find the best connection between two places. Another very common usage is showing all departures from a given stop, which is an advantage for waiting passengers especially in urban transport (where both intervals and distances are lower).

1.6 Chapters overview

In chapter 2, we show data sources we use for our application and tools for working with them. Following the chapter 3, we briefly mention other software solutions tied to the functionalities we use. In chapter 4, we show the functional requirements for TTP. In 5, we elaborate on how the SDBS and MDBS solutions are implemented. The chapter 6, shows how is TTP integrated with map data. The chapter 7, analyzes TTP's performance on some real-life networks. In chapter 8, we show the general design of TTP, serving also as user manual. A chapter 9 shows how to install the application for user's own experiments. In chapter 10, we show the programmer documentation for TTP. Finally, we present conclusion in 11

[5] https://portal.cisjr.cz

[6] ftp://ftp.cisjr.cz/JDF

[7] https://www.google.com/maps

Chapter 2

Used data sources

This chapter describes both external an internal data sources used in TTP.

2.1 Programming tools used in TTP

The main programming language used in TTP is Python 3.10 [9]. It is a high-level language with a wide range of libraries available, which makes it a good choice for data processing and visualization.

For front-end, we use Flask [10] as a web framework, which utilizes HTML5 for the user interface. Other libraries are mentioned in the text where they are relevant.

2.2 Definitions of timetable terms

Let us define some terms used in the timetable context.

1. Stop: A place for embarking and disembarking of passengers. Identified by name (in TTP, all stops are considered to have one location only).

2. Route: An ordered collection of stops around which a bus passes by.

3. Serving a stop: Bus passing by a stop while allowing passengers to embark and disembark.

4. Trip: A description of a route with given times during which each stop must be served. A trip is identified by its number and a line (see later) it belongs to.

5. Stop event: A function of trip and stop, the "time" displayed on a timetable, commonly meant as "departure" and "arrival".

6. Transport agency: A company responsible for driving the buses on specified trips.
7. Line: A collection of trips. Usually, most trips following the same line also have similar route, but this is not always the case.
8. Timetable: A textual representation of a line (in our cases, one timetable always contain only one line) containing stops, trips, departure/arrival times and other data relevant to passengers.
9. Chaining of trips: Two trips served one after another by a bus.
10. Deadhead: The act of a bus driving between chained trips (from last stop of an earlier trip to a first stop of later trip). We also include zero length deadheads (turnarounds).
11. Bus block: A collection of trips served by one bus.

2.3 Timetables in JDF

The format for persistent data storage is JDF 1.11 (Jednotný datový formát (Unified Data Format)) as described by Czech Ministry of Transport [11] . All public bus lines currently operating in the Czech Republic have to be published in this format. They can be accessed freely at [12]. This gives us a good source for testing data, and also somewhat standardized format at national level.

2.3.1 Description of the JDF format

The full description is available at [11]. This paragraph only describes the essential things and things which might not be obvious at the first glance.

1. One JDF "batch" is a folder containing .txt files at the top level. the name of folder is irrelevant (the public data source, after unzipping, uses one folder per line, and the folders use integer names, starting with 1.).
2. The files have .txt extension, but are in fact CSV files (no headers, cp-1252 encoding, semicolon delimiter, mandatory quotes)
3. There is no file (neither mandatory nor optional) which would describe the location of the stops. This is due to the fact that these data have to only be equivalent to paper timetables (shown at bus stops) where this info is not necessary either. In contrary, the alternative GTFS format requires these [13].

4. Despite the specification stating that number of a stop must be taken from CIS JŘ registry (basically a nation-wide database of stops), this is not true for actually published JDF batches (where the number must be only unique within the batch, i.e. a primary key).

2.3.2 Stop name standardization

The most common issue while working with the JDF input data was the standardization of stop names.

A standard stop name, as displayed on timetable, should have 3 parts: city (mandatory), city part, and location. This means the stop name on timetable can be in the format “[city],[city part],[location]“ with trailing commas removed.

Since city names can also be ambiguous (mostly occurs in case of small towns like “Lhota“), the actual stop name also contains field “BlizkaObec“ (nearby city), which we represent in TTP in brackets after the stop name.

Finally, a stop can be also identified by a country it belongs to, which we do not display in our output data, due to simplicity.

Some published JDF data save on the stop names by omitting the city name. This is mostly done for the convenience of users who do not need to see repeated city name in case of urban transport. On the other hand, this can make data processing harder if these stops are also used by regional transport which would often use the full name.

More technically evolved systems simply solve this by storing the stop names by their full name (in official JDF data), but having a specific mapping of the stop names when displaying them to the users, on custom timetables, and bus information systems. An easy example is Prague, where all stops are named “Praha„[stop name]“ , but in all data for users and inside bus information system, only “[stop name]“ is used. Adding a city part makes this even harder, as non-locals might not differentiate between sovereign city and city part. Finally, information systems rarely display double commas in stop names.

We implement a solution to this (as pre-processing step) in the script `fix_stop_names.py`, which requires some external knowledge of the region and municipalities from the user.

This script attempts to fix all of these:

- Fixing commas within names - e.g.“Ostrava,Poruba“ as city becomes “Ostrava“ as city and “Poruba“ as city part.

- Central city name - e.g. “Zlín“: this is appended to a stop name if it is not already present.

- Independent city names - e.g. "Želechovice"; "Otrokovice": central city name is not appended to these.

- Option for moving location to the third part of the name - e.g. "Praha,Černý Most" becomes "Praha,,Černý Most." This can generate false positives if the actual city name contains of two parts and location is not present, which are best to fix manually (e.g. "Uherský Brod,Újezdec").

2.4 Human-readable timetable format

In the Czech Republic, the timetables are available at www.portal.cisjr.cz in PDF. The public transport agencies use similar format for printed timetables at the bus stops with some changes like separating workdays and weekends or inserting more rows for transfer options. We can see the comparison on image 2.1.

This format is easy to read, but harder to edit or parse by machines. This is why we decided to use another format, XLSX ([14]) which is easy to edit with spreadsheet editors such as LibreOfficeCalc[1] or Apache OpenOffice[2]. and there also exist open tools for programmatic editing of XLSX files, such as `openpyxl`[3].

Such conversion application was already made by Papež in [15], using the official data as a source and providing richer user interface. An example can be seen on image 2.2.

Our method for converting JDF to XLSX was written from scratch, giving similar output, but focusing on the ability to re-import the data back to TTP.

2.4.1 Export of timetable as XLSX

The advantages of exporting the timetables as XLSX are as following: First, the data, when opened in a spreadsheet editor such as Microsoft Excel, look similar way to the official PDF renderings. Second, the data can be directly manipulated. As spreadsheet editors support operations of inserting rows or columns, adding additional trips or stops becomes easier.

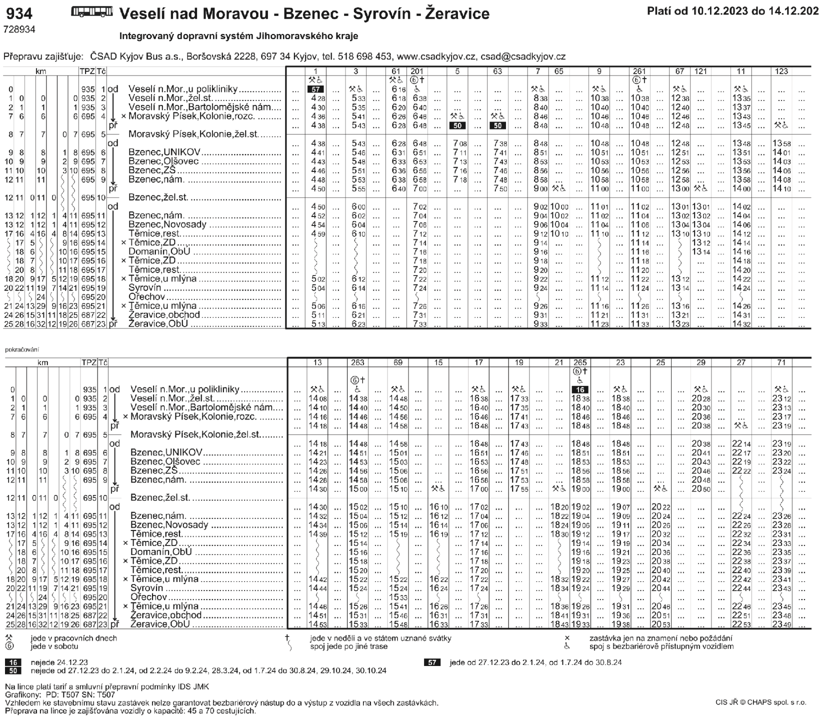
The editing part and the whole format is described in the user documentation (8.7). The export itself is not meant only for editing purposes, but also to verify the data on timetable. It must be noted that the exported data do not encompass all information available in the original JDF batch.

We can call this format TT-XLSX in the rest of the text.

[1] https://www.libreoffice.org/discover/calc/

[2] https://www.openoffice.org/

[3] https://openpyxl.readthedocs.io/en/stable/



934 🚌 **Veselí nad Moravou - Bzenec - Syrovín - Žeravice** **Platí od 10.12.2023 do 14.12.2024**

728934

Integrovaný dopravní systém Jihomoravského kraje

Přepravu zajišťuje: ČSAD Kyjov Bus a.s., Boršovská 2228, 697 34 Kyjov, tel. 518 698 453, www.csadkyjov.cz, csad@csadkyjov.cz

km							TPZ	Tč			1	3	61	201	5	63	7	65	9	261	67	121	11	123
											⚒♿		⚒♿	⑥†						⑥†				
0							935	1	od	Veselí n.Mor.,u polikliniky	57	⚒♿	6 16	♿	...	...	⚒♿	...	⚒♿	♿	⚒♿	...	⚒♿	...
1	0		0			0	935	2	↓	Veselí n.Mor.,žel.st.	4 28	5 33	6 18	6 38	...	...	8 38	...	10 38	10 38	12 38	...	13 35	...
2	1		1			1	935	3		Veselí n.Mor.,Bartolomějské nám.	4 30	5 35	6 20	6 40	...	...	8 40	...	10 40	10 40	12 40	...	13 37	...
7	6		6			6	695	4		× Moravský Písek,Kolonie,rozc.	4 36	5 41	6 26	6 46	⚒♿	⚒♿	8 46	...	10 46	10 46	12 46	...	13 43	...
									př		4 38	5 43	6 28	6 48	50	50	8 48	...	10 48	10 48	12 48	...	13 45	⚒♿
8	7		7		0	7	695	5		Moravský Písek,Kolonie,žel.st.														
									od		4 38	5 43	6 28	6 48	7 08	7 38	8 48	...	10 48	10 48	12 48	...	13 48	13 58
9	8		8		1	8	695	6		Bzenec,UNIKOV	4 41	5 46	6 31	6 51	7 11	7 41	8 51	...	10 51	10 51	12 51	...	13 51	14 01
10	9		9		2	9	695	7		Bzenec,Olšovec	4 43	5 48	6 33	6 53	7 13	7 43	8 53	...	10 53	10 53	12 53	...	13 53	14 03
11	10		10		3	10	695	8		Bzenec,ZŠ	4 46	5 51	6 36	6 56	7 16	7 46	8 56	...	10 56	10 56	12 56	...	13 56	14 06
12	11		11		≀	≀	695	9		Bzenec,nám.	4 48	5 53	6 38	6 58	7 18	7 48	8 58	...	10 58	10 58	12 58	...	13 58	14 08
									př		4 50	5 55	6 40	7 00	...	7 50	9 00	⚒♿	11 00	11 00	13 00	⚒♿	14 00	14 10
12	11	0	11	0	≀	≀	695	10		Bzenec,žel.st.														
									od		4 50	6 00	...	7 02	...	...	9 02	10 00	11 01	11 02	13 01	13 01	14 02	...
13	12	1	12	1	4	11	695	11		Bzenec,nám.	4 52	6 02	...	7 04	...	...	9 04	10 02	11 02	11 04	13 02	13 02	14 04	...
13	12	1	12	1	4	11	695	12		Bzenec,Novosady	4 54	6 04	...	7 06	...	...	9 06	10 04	11 04	11 06	13 04	13 04	14 06	...
17	16	4	16	4	8	14	695	13		Těmice,rest.	4 59	6 10	...	7 12	...	...	9 12	10 10	11 10	11 12	13 10	13 10	14 12	...
≀	17	5	≀	≀	9	16	695	14		× Těmice,ZD	≀	≀	...	7 14	...	...	9 14	...	≀	11 14	≀	13 12	14 14	...
≀	18	6	≀	≀	10	16	695	15		Domanín,ObÚ	≀	≀	...	7 16	...	...	9 16	...	≀	11 16	≀	13 14	14 16	...
≀	18	7	≀	≀	10	17	695	16		× Těmice,ZD	≀	≀	...	7 18	...	...	9 18	...	≀	11 18	≀	...	14 18	...
≀	20	8	≀	≀	11	18	695	17		Těmice,rest.	≀	≀	...	7 20	...	...	9 20	...	≀	11 20	≀	...	14 20	...
18	20	9	17	5	12	19	695	18		× Těmice,u mlýna	5 02	6 12	...	7 22	...	...	9 22	...	11 12	11 22	13 12	...	14 22	...
20	22	11	19	7	14	21	695	19		Syrovín	5 04	6 14	...	7 24	...	...	9 24	...	11 14	11 24	13 14	...	14 24	...
≀	≀	≀	24	≀	≀	≀	695	20		Ořechov	≀	≀	...	≀	...	...	≀	...	≀	≀	≀	...	≀	...
21	24	13	29	9	16	23	695	21		× Těmice,u mlýna	5 06	6 16	...	7 26	...	...	9 26	...	11 16	11 26	13 16	...	14 26	...
24	26	15	31	11	18	25	687	22	↓	Žeravice,obchod	5 11	6 21	...	7 31	...	...	9 31	...	11 21	11 31	13 21	...	14 31	...
25	28	16	32	12	19	26	687	23	př	Žeravice,ObÚ	5 13	6 23	...	7 33	...	...	9 33	...	11 23	11 33	13 23	...	14 32	...

pokračování

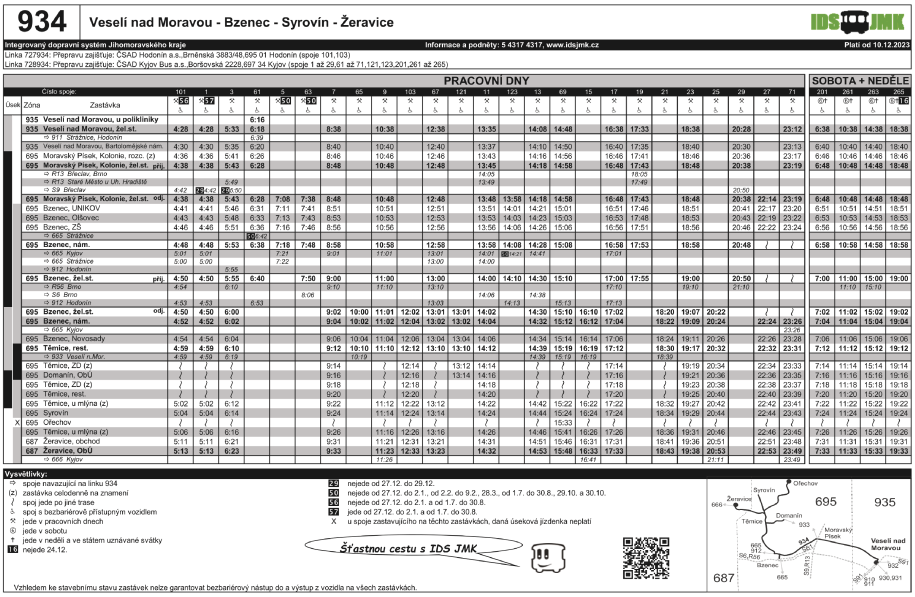
km							TPZ	Tč			13	263	69	15	17	19	21	265	23	25	29	27	71
																		⑥†					
												⑥†						♿					
0							935	1	od	Veselí n.Mor.,u polikliniky	⚒♿	♿	⚒♿	...	⚒♿	⚒♿	...	16	⚒♿	...	⚒♿	...	⚒♿
1	0		0			0	935	2	↓	Veselí n.Mor.,žel.st.	14 08	14 38	14 48	...	16 38	17 33	...	18 38	18 38	...	20 28	...	23 12
2	1		1			1	935	3		Veselí n.Mor.,Bartolomějské nám.	14 10	14 40	14 50	...	16 40	17 35	...	18 40	18 40	...	20 30	...	23 13
7	6		6			6	695	4		× Moravský Písek,Kolonie,rozc.	14 16	14 46	14 56	...	16 46	17 41	...	18 46	18 46	...	20 36	...	23 17
									př		14 18	14 48	14 58	...	16 48	17 43	...	18 48	18 48	...	20 38	⚒♿	23 19
8	7		7		0	7	695	5		Moravský Písek,Kolonie,žel.st.													
									od		14 18	14 48	14 58	...	16 48	17 43	...	18 48	18 48	...	20 38	22 14	23 19
9	8		8		1	8	695	6		Bzenec,UNIKOV	14 21	14 51	15 01	...	16 51	17 46	...	18 51	18 51	...	20 41	22 17	23 20
10	9		9		2	9	695	7		Bzenec,Olšovec	14 23	14 53	15 03	...	16 53	17 48	...	18 53	18 53	...	20 43	22 19	23 22
11	10		10		3	10	695	8		Bzenec,ZŠ	14 26	14 56	15 06	...	16 56	17 51	...	18 56	18 56	...	20 46	22 22	23 24
12	11		11		≀	≀	695	9		Bzenec,nám.	14 28	14 58	15 08	...	16 58	17 53	...	18 58	18 58	...	20 48	≀	≀
									př		14 30	15 00	15 10	⚒♿	17 00	17 55	⚒♿	19 00	19 00	⚒♿	20 50	≀	≀
12	11	0	11	0	≀	≀	695	10		Bzenec,žel.st.													
									od		14 30	15 02	15 10	16 10	17 02	...	18 20	19 02	19 07	20 22	...	≀	≀
13	12	1	12	1	4	11	695	11		Bzenec,nám.	14 32	15 04	15 12	16 12	17 04	...	18 22	19 04	19 09	20 24	...	22 24	23 26
13	12	1	12	1	4	11	695	12		Bzenec,Novosady	14 34	15 06	15 14	16 14	17 06	...	18 24	19 06	19 11	20 26	...	22 26	23 28
17	16	4	16	4	8	14	695	13		Těmice,rest.	14 39	15 12	15 19	16 19	17 12	...	18 30	19 12	19 17	20 32	...	22 32	23 31
≀	17	5	≀	≀	9	16	695	14		× Těmice,ZD	≀	15 14	≀	≀	17 14	...	≀	19 14	19 19	20 34	...	22 34	23 33
≀	18	6	≀	≀	10	16	695	15		Domanín,ObÚ	≀	15 16	≀	≀	17 16	...	≀	19 16	19 21	20 36	...	22 36	23 35
≀	18	7	≀	≀	10	17	695	16		× Těmice,ZD	≀	15 18	≀	≀	17 18	...	≀	19 18	19 23	20 38	...	22 38	23 37
≀	20	8	≀	≀	11	18	695	17		Těmice,rest.	≀	15 20	≀	≀	17 20	...	≀	19 20	19 25	20 40	...	22 40	23 39
18	20	9	17	5	12	19	695	18		× Těmice,u mlýna	14 42	15 22	15 22	16 22	17 22	...	18 32	19 22	19 27	20 42	...	22 42	23 41
20	22	11	19	7	14	21	695	19		Syrovín	14 44	15 24	15 24	16 24	17 24	...	18 34	19 24	19 29	20 44	...	22 44	23 43
≀	≀	≀	24	≀	≀	≀	695	20		Ořechov	≀	≀	15 33	≀	≀	...	≀	≀	≀	≀	...	≀	≀
21	24	13	29	9	16	23	695	21		× Těmice,u mlýna	14 46	15 26	15 41	16 26	17 26	...	18 36	19 26	19 31	20 46	...	22 46	23 45
24	26	15	31	11	18	25	687	22	↓	Žeravice,obchod	14 51	15 31	15 46	16 31	17 31	...	18 41	19 31	19 36	20 51	...	22 51	23 48
25	28	16	32	12	19	26	687	23	př	Žeravice,ObÚ	14 53	15 33	15 48	16 33	17 33	...	18 43	19 33	19 38	20 53	...	22 53	23 49

⚒ jede v pracovních dnech
⑥ jede v sobotu
† jede v neděli a ve státem uznané svátky
≀ spoj jede po jiné trase
× zastávka jen na znamení nebo požádání
♿ spoj s bezbariérově přístupným vozidlem

16 nejede 24.12.23
50 nejede od 27.12.23 do 2.1.24, od 2.2.24 do 9.2.24, 28.3.24, od 1.7.24 do 30.8.24, 29.10.24, 30.10.24
57 jede od 27.12.23 do 2.1.24, od 1.7.24 do 30.8.24

Na lince platí tarif a smluvní přepravní podmínky IDS JMK
Grafikony: PD: T507 SN: T507
Vzhledem ke stavebnímu stavu zastávek nelze garantovat bezbariérový nástup do a výstup z vozidla na všech zastávkách.
Přeprava na lince je zajišťována vozidly o kapacitě: 45 a 70 cestujících.

CIS JŘ © CHAPS spol. s r.o.



934 **Veselí nad Moravou - Bzenec - Syrovín - Žeravice** IDS JMK

Integrovaný dopravní systém Jihomoravského kraje **Informace a podněty: 5 4317 4317, www.idsjmk.cz** **Platí od 10.12.2023**

Linka 727934: Přepravu zajišťuje: ČSAD Hodonín a.s.,Brněnská 3883/48,695 01 Hodonín (spoje 101,103)
Linka 728934: Přepravu zajišťuje: ČSAD Kyjov Bus a.s.,Boršovská 2228,697 34 Kyjov (spoje 1 až 29,61 až 71,121,123,201,261 až 265)

			PRACOVNÍ DNY																								SOBOTA + NEDĚLE			
		Číslo spoje:	101	1	3	61	5	63	7	65	9	103	67	121	11	123	13	69	15	17	19	21	23	25	29	27	71	201	261	263	265
Úsek	Zóna	Zastávka	⚒ 56 ♿	⚒ 57 ♿	⚒ ♿	⚒ ♿	⚒ 50 ♿	⚒ 50 ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⚒ ♿	⑥† ♿	⑥† ♿	⑥† ♿	⑥† 16 ♿
	935	**Veselí nad Moravou, u polikliniky**				**6:16**																									
	935	**Veselí nad Moravou, žel.st.**	**4:28**	**4:28**	**5:33**	**6:18**			**8:38**		**10:38**		**12:38**		**13:35**		**14:08**	**14:48**		**16:38**	**17:33**		**18:38**		**20:28**		**23:12**	**6:38**	**10:38**	**14:38**	**18:38**
		⇨ *911 Strážnice, Hodonín*				*6:39*																									
	935	Veselí nad Moravou, Bartolomějské nám.	4:30	4:30	5:35	6:20			8:40		10:40		12:40		13:37		14:10	14:50		16:40	17:35		18:40		20:30		23:13	6:40	10:40	14:40	18:40
	695	Moravský Písek, Kolonie, rozc. (z)	4:36	4:36	5:41	6:26			8:46		10:46		12:46		13:43		14:16	14:56		16:46	17:41		18:46		20:36		23:17	6:46	10:46	14:46	18:46
	695	**Moravský Písek, Kolonie, žel.st.** přij.	**4:38**	**4:38**	**5:43**	**6:28**			**8:48**		**10:48**		**12:48**		**13:45**		**14:18**	**14:58**		**16:48**	**17:43**		**18:48**		**20:38**		**23:19**	**6:48**	**10:48**	**14:48**	**18:48**
		⇨ *R13 Břeclav, Brno*													*14:05*						*18:05*										
		⇨ *R13 Staré Město u Uh. Hradiště*			*5:49*										*13:49*						*17:49*										
		⇨ *S9 Břeclav*	*4:42*	29 *4:42*	29 *5:50*																				*20:50*						
	695	**Moravský Písek, Kolonie, žel.st.** odj.	**4:38**	**4:38**	**5:43**	**6:28**	**7:08**	**7:38**	**8:48**		**10:48**		**12:48**		**13:48**	**13:58**	**14:18**	**14:58**		**16:48**	**17:43**		**18:48**		**20:38**	**22:14**	**23:19**	**6:48**	**10:48**	**14:48**	**18:48**
	695	Bzenec, UNIKOV	4:41	4:41	5:46	6:31	7:11	7:41	8:51		10:51		12:51		13:51	14:01	14:21	15:01		16:51	17:46		18:51		20:41	22:17	23:20	6:51	10:51	14:51	18:51
	695	Bzenec, Olšovec	4:43	4:43	5:48	6:33	7:13	7:43	8:53		10:53		12:53		13:53	14:03	14:23	15:03		16:53	17:48		18:53		20:43	22:19	23:22	6:53	10:53	14:53	18:53
	695	Bzenec, ZŠ	4:46	4:46	5:51	6:36	7:16	7:46	8:56		10:56		12:56		13:56	14:06	14:26	15:06		16:56	17:51		18:56		20:46	22:22	23:24	6:56	10:56	14:56	18:56
		⇨ *665 Strážnice*				50 *6:42*																									
	695	**Bzenec, nám.**	**4:48**	**4:48**	**5:53**	**6:38**	**7:18**	**7:48**	**8:58**		**10:58**		**12:58**		**13:58**	**14:08**	**14:28**	**15:08**		**16:58**	**17:53**		**18:58**		**20:48**	≀	≀	**6:58**	**10:58**	**14:58**	**18:58**
		⇨ *665 Kyjov*	*5:01*	*5:01*			*7:21*		*9:01*		*11:01*		*13:01*		*14:01*	50 *14:21*	*14:41*			*17:01*											
		⇨ *665 Strážnice*	*5:00*	*5:00*			*7:22*						*13:00*		*14:00*																
		⇨ *912 Hodonín*			*5:55*																										
	695	**Bzenec, žel.st.** přij.	**4:50**	**4:50**	**5:55**	**6:40**		**7:50**	**9:00**		**11:00**		**13:00**		**14:00**	**14:10**	**14:30**	**15:10**		**17:00**	**17:55**		**19:00**		**20:50**	≀	≀	**7:00**	**11:00**	**15:00**	**19:00**
		⇨ *R56 Brno*	*4:54*		*6:10*				*9:10*		*11:10*		*13:10*							*17:10*			*19:10*		*21:10*				*11:10*	*15:10*	
		⇨ *S6 Brno*						*8:06*							*14:06*		*14:38*														
		⇨ *912 Hodonín*	*4:53*	*4:53*		*6:53*							*13:03*			*14:13*		*15:13*		*17:13*											
	695	**Bzenec, žel.st.** odj.	**4:50**	**4:50**	**6:00**				**9:02**	**10:00**	**11:01**	**12:02**	**13:01**	**13:01**	**14:02**		**14:30**	**15:10**	**16:10**	**17:02**		**18:20**	**19:07**	**20:22**		≀	≀	**7:02**	**11:02**	**15:02**	**19:02**
	695	**Bzenec, nám.**	**4:52**	**4:52**	**6:02**				**9:04**	**10:02**	**11:02**	**12:04**	**13:02**	**13:02**	**14:04**		**14:32**	**15:12**	**16:12**	**17:04**		**18:22**	**19:09**	**20:24**		**22:24**	**23:26**	**7:04**	**11:04**	**15:04**	**19:04**
		⇨ *665 Kyjov*																									*23:26*				
	695	Bzenec, Novosady	4:54	4:54	6:04				9:06	10:04	11:04	12:06	13:04	13:04	14:06		14:34	15:14	16:14	17:06		18:24	19:11	20:26		22:26	23:28	7:06	11:06	15:06	19:06
	695	**Těmice, rest.**	**4:59**	**4:59**	**6:10**				**9:12**	**10:10**	**11:10**	**12:12**	**13:10**	**13:10**	**14:12**		**14:39**	**15:19**	**16:19**	**17:12**		**18:30**	**19:17**	**20:32**		**22:32**	**23:31**	**7:12**	**11:12**	**15:12**	**19:12**
		⇨ *933 Veselí n.Mor.*	*4:59*	*4:59*	*6:19*					*10:19*							*14:39*	*15:19*	*16:19*			*18:39*									
	695	Těmice, ZD (z)	≀	≀	≀				9:14		≀	12:14	≀	13:12	14:14		≀	≀	≀	17:14		≀	19:19	20:34		22:34	23:33	7:14	11:14	15:14	19:14
	695	Domanín, ObÚ	≀	≀	≀				9:16		≀	12:16	≀	13:14	14:16		≀	≀	≀	17:16		≀	19:21	20:36		22:36	23:35	7:16	11:16	15:16	19:16
	695	Těmice, ZD (z)	≀	≀	≀				9:18		≀	12:18	≀		14:18		≀	≀	≀	17:18		≀	19:23	20:38		22:38	23:37	7:18	11:18	15:18	19:18
	695	Těmice, rest.	≀	≀	≀				9:20		≀	12:20	≀		14:20		≀	≀	≀	17:20		≀	19:25	20:40		22:40	23:39	7:20	11:20	15:20	19:20
	695	Těmice, u mlýna (z)	5:02	5:02	6:12				9:22		11:12	12:22	13:12		14:22		14:42	15:22	16:22	17:22		18:32	19:27	20:42		22:42	23:41	7:22	11:22	15:22	19:22
	695	Syrovín	5:04	5:04	6:14				9:24		11:14	12:24	13:14		14:24		14:44	15:24	16:24	17:24		18:34	19:29	20:44		22:44	23:43	7:24	11:24	15:24	19:24
X	695	Ořechov	≀	≀	≀				≀		≀	≀	≀		≀		≀	15:33	≀	≀		≀	≀	≀		≀	≀	≀	≀	≀	≀
	695	Těmice, u mlýna (z)	5:06	5:06	6:16				9:26		11:16	12:26	13:16		14:26		14:46	15:41	16:26	17:26		18:36	19:31	20:46		22:46	23:45	7:26	11:26	15:26	19:26
	687	Žeravice, obchod	5:11	5:11	6:21				9:31		11:21	12:31	13:21		14:31		14:51	15:46	16:31	17:31		18:41	19:36	20:51		22:51	23:48	7:31	11:31	15:31	19:31
	687	**Žeravice, ObÚ**	**5:13**	**5:13**	**6:23**				**9:33**		**11:23**	**12:33**	**13:23**		**14:32**		**14:53**	**15:48**	**16:33**	**17:33**		**18:43**	**19:38**	**20:53**		**22:53**	**23:49**	**7:33**	**11:33**	**15:33**	**19:33**
		⇨ *666 Kyjov*									*11:26*								*16:41*					*21:11*			*23:49*				

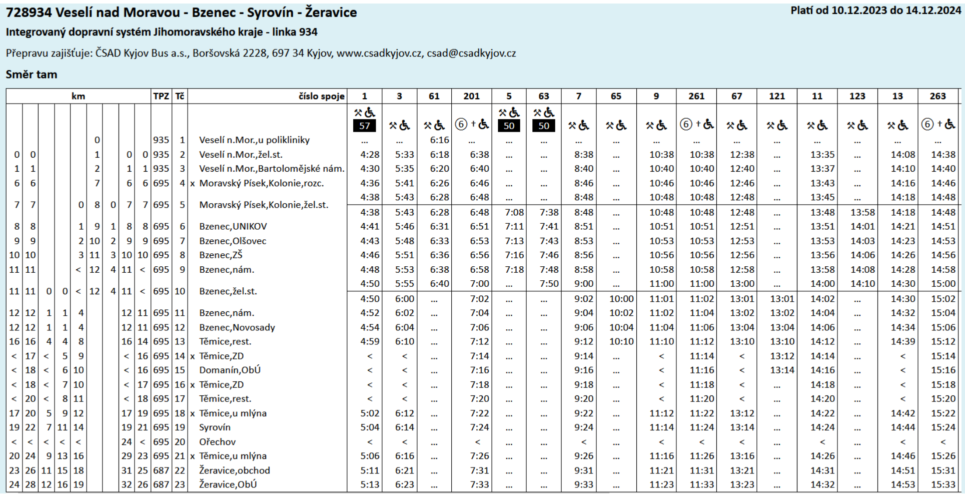
Vysvětlivky:

⇨ spoje navazující na linku 934
(z) zastávka celodenně na znamení
≀ spoj jede po jiné trase
♿ spoj s bezbariérově přístupným vozidlem
⚒ jede v pracovních dnech
⑥ jede v sobotu
† jede v neděli a ve státem uznávané svátky
16 nejede 24.12.

29 nejede od 27.12. do 29.12.
50 nejede od 27.12. do 2.1., od 2.2. do 9.2., 28.3., od 1.7. do 30.8., 29.10. a 30.10.
56 nejede od 27.12. do 2.1. a od 1.7. do 30.8.
57 jede od 27.12. do 2.1. a od 1.7. do 30.8.
X u spoje zastavujícího na těchto zastávkách, daná úseková jízdenka neplatí

Vzhledem ke stavebnímu stavu zastávek nelze garantovat bezbariérový nástup do a výstup z vozidla na všech zastávkách.

Figure 2.1 A part of timetable of line 728934 in CIS JŘ, and as styled by the transport organizer. Sources: Portál CIS JŘ and IDS JMK



728934 Veselí nad Moravou - Bzenec - Syrovín - Žeravice **Platí od 10.12.2023 do 14.12.2024**

Integrovaný dopravní systém Jihomoravského kraje - linka 934

Přepravu zajišťuje: ČSAD Kyjov Bus a.s., Boršovská 2228, 697 34 Kyjov, www.csadkyjov.cz, csad@csadkyjov.cz

Směr tam

km									TPZ	Tč		číslo spoje	1	3	61	201	5	63	7	65	9	261	67	121	11	123	13	263
													⚒♿ 57	⚒♿	⚒♿	⑥ † ♿	⚒♿ 50	⚒♿ 50	⚒♿	⚒♿	⚒♿	⑥ † ♿	⚒♿	⚒♿	⚒♿	⚒♿	⚒♿	⑥ † ♿
					0				935	1		Veselí n.Mor.,u polikliniky	...	...	6:16	...	...	...	...	...	...	...	...	...	...	...	...	...
0	0				1		0	0	935	2		Veselí n.Mor.,žel.st.	4:28	5:33	6:18	6:38	...	...	8:38	...	10:38	10:38	12:38	...	13:35	...	14:08	14:38
1	1				2		1	1	935	3		Veselí n.Mor.,Bartolomějské nám.	4:30	5:35	6:20	6:40	...	...	8:40	...	10:40	10:40	12:40	...	13:37	...	14:10	14:40
6	6				7		6	6	695	4	x	Moravský Písek,Kolonie,rozc.	4:36	5:41	6:26	6:46	...	...	8:46	...	10:46	10:46	12:46	...	13:43	...	14:16	14:46
7	7			0	8	0	7	7	695	5		Moravský Písek,Kolonie,žel.st.	4:38	5:43	6:28	6:48	...	...	8:48	...	10:48	10:48	12:48	...	13:45	...	14:18	14:48
													4:38	5:43	6:28	6:48	7:08	7:38	8:48	...	10:48	10:48	12:48	...	13:48	13:58	14:18	14:48
8	8			1	9	1	8	8	695	6		Bzenec,UNIKOV	4:41	5:46	6:31	6:51	7:11	7:41	8:51	...	10:51	10:51	12:51	...	13:51	14:01	14:21	14:51
9	9			2	10	2	9	9	695	7		Bzenec,Olšovec	4:43	5:48	6:33	6:53	7:13	7:43	8:53	...	10:53	10:53	12:53	...	13:53	14:03	14:23	14:53
10	10			3	11	3	10	10	695	8		Bzenec,ZŠ	4:46	5:51	6:36	6:56	7:16	7:46	8:56	...	10:56	10:56	12:56	...	13:56	14:06	14:26	14:56
11	11			<	12	4	11	<	695	9		Bzenec,nám.	4:48	5:53	6:38	6:58	7:18	7:48	8:58	...	10:58	10:58	12:58	...	13:58	14:08	14:28	14:58
11	11	0	0	<	12	4	11	<	695	10		Bzenec,žel.st.	4:50	5:55	6:40	7:00	...	7:50	9:00	...	11:00	11:00	13:00	...	14:00	14:10	14:30	15:00
													4:50	6:00	...	7:02	...	...	9:02	10:00	11:01	11:02	13:01	13:01	14:02	...	14:30	15:02
12	12	1	1	4			12	11	695	11		Bzenec,nám.	4:52	6:02	...	7:04	...	...	9:04	10:02	11:02	11:04	13:02	13:02	14:04	...	14:32	15:04
12	12	1	1	4			12	11	695	12		Bzenec,Novosady	4:54	6:04	...	7:06	...	...	9:06	10:04	11:04	11:06	13:04	13:04	14:06	...	14:34	15:06
16	16	4	4	8			16	14	695	13		Těmice,rest.	4:59	6:10	...	7:12	...	...	9:12	10:10	11:10	11:12	13:10	13:10	14:12	...	14:39	15:12
<	17	<	5	9			<	16	695	14	x	Těmice,ZD	<	<	...	7:14	...	...	9:14	...	<	11:14	<	13:12	14:14	...	<	15:14
<	18	<	6	10			<	16	695	15		Domanín,ObÚ	<	<	...	7:16	...	...	9:16	...	<	11:16	<	13:14	14:16	...	<	15:16
<	18	<	7	10			<	17	695	16	x	Těmice,ZD	<	<	...	7:18	...	...	9:18	...	<	11:18	<	...	14:18	...	<	15:18
<	20	<	8	11			<	18	695	17		Těmice,rest.	<	<	...	7:20	...	...	9:20	...	<	11:20	<	...	14:20	...	<	15:20
17	20	5	9	12			17	19	695	18	x	Těmice,u mlýna	5:02	6:12	...	7:22	...	...	9:22	...	11:12	11:22	13:12	...	14:22	...	14:42	15:22
19	22	7	11	14			19	21	695	19		Syrovín	5:04	6:14	...	7:24	...	...	9:24	...	11:14	11:24	13:14	...	14:24	...	14:44	15:24
<	<	<	<	<			24	<	695	20		Ořechov	<	<	...	<	...	...	<	...	<	<	<	...	<	...	<	<
20	24	9	13	16			29	23	695	21	x	Těmice,u mlýna	5:06	6:16	...	7:26	...	...	9:26	...	11:16	11:26	13:16	...	14:26	...	14:46	15:26
23	26	11	15	18			31	25	687	22		Žeravice,obchod	5:11	6:21	...	7:31	...	...	9:31	...	11:21	11:31	13:21	...	14:31	...	14:51	15:31
24	28	12	16	19			32	26	687	23		Žeravice,ObÚ	5:13	6:23	...	7:33	...	...	9:33	...	11:23	11:33	13:23	...	14:32	...	14:53	15:33

Figure 2.2 A part of timetable of line 728934, as seen on web portal by Radek Papež. Source: portal.radekpapez.cz

Unlike JDF, where data are split into individual files by their type (line,trip...), the TT-XLSX splits the worksheets by line and direction, as can be seen on official timetables.

2.4.2 Upload of timetable in XLSX format

Due to the tabular nature of timetable, XLSX format (or any spreadsheet format in general) is a good way to present timetable proposal to authorities. The disadvantage is that there is apparently no specialized tool to convert XLSX data into JDF, so it was created using the same principles as the JDF to TT-XLSX conversion.

The TT-XLSX is designed in such a way the timetable can be exported in such a way the user can immediately re-import it to our TTP.

This causes some data losses, as not all information would properly fit on the XLSX sheet, and general parsing would be hard enough, so we decided to properly convert only the basic data - stops, trips and lines, which are the most important for the scheduling calculation.

The only thing which was omitted and which changes our scheduling calculation result are time codes - signs listing specific dates when a trip operates or not. However, periodic signs (whether trip operates on workdays, weekends, Mondays) are still parsed.

2.5 OpenStreetMap

The OpenStreetMap is our choice for analyzing travel times and drawing trips. Its public specification is supported by many programming libraries. As the program is written in Python, the libraries which we use are *osmnx*[4], *leaflet*[5], and a modified version of *mplleaflet*[6]. Its disadvantage is the incompleteness of data, making some lines be drawn poorly or to miscalculate driving times.

2.6 Merging of JDF timetables

The publicly available JDF batches are saved in the format "one batch - one line". However, most public transport systems are organized by multiple lines, where one bus can serve multiple trips belonging to different lines. Therefore, TTP is designed to merge multiple timetables into one JDF batch with unique identifier. This procedure is rather easy, as most data (e.g. trip metadata) are disjoint. The major exceptions are the code map and the names of stops, which are handled specifically. It is assumed that the merged timetables belong to the same transit agency (or at least organizer) who follows the same conventions such as stop naming in all their timetables. In fact, the only observed discrepancy were equal "global stop attributes", such as whether a stop follows barrier-free access, which is irrelevant for our problem.

As the stops are contained in only two files (Zastavky.txt and Zaslinky.txt), if the merging is not perfect (such as if the user failed to standardize the stop names), rewriting respective record in Zaslinky.txt (and deleting record from Zastavky.txt, whose identifier is renamed) is enough.

2.7 Stop editing

The knowledge of stop locations is crucial for any transport planning, with visualization of the routes being the main motivation. The issue is, JDF does not require nor allow this. One of the reasons might be, that a timetable row (in the file "Zaslinky.txt") can represent stop on multiple routes, which can use different platforms. The most common example are reverse routes, which increases the amount of needed platforms to two. We decide on a solution that to one stop, one location is assigned, for example an average of all its platforms' locations - this is how Mapy.cz [16] displays stops at far zoom. The list of stops with known

[4] https://osmnx.readthedocs.io/en/stable/

[5] https://leafletjs.com/

[6] https://github.com/jwass/mplleaflet

locations is maintained in TTP, outside the uploaded JDF batches. As we know, JDF does not enforce global stop identifiers by number, therefore each stop is identified by its name. And the final issue is, how to get the actual data - how to assign a stop name to a location? This get mentioned later, for short we can just say that TTP does not use any automated methods - the data have to be uploaded from external source, or the stops can be added manually with graphical user interface (with the help of a map background).

2.8 Scheduling calculation

Knowing which bus to assign to each individual trip is the core of organizing any public transport. TTP takes a very simplistic take on the problematic; it tries to satisfy the assignment objective on two levels:

1. minimizing the bus count
2. minimizing the deadhead (including pull-in and pull-out) times

This is rather an expected objective, which minimizes most of the operational costs. More is explained in chapter 5. What is not considered, are the drivers operating the buses and their breaks - required (by law), which would make the scheduling illegal if ignored, and also involuntary breaks (too long downtime serving only morning and afternoon trips), which also incurs rather high operational costs.

Therefore, the simple calculation serves only as a demonstrative example, but adding more complicated calculation should be reasonably easy within the source code.

Chapter 3

Alternative solutions and software

3.1 Timetable formats for public data exchange

In Czech Republic, the JDF 1.11 can be considered as a standard for bus transport timetables. It should be mentioned that transport agencies themselves (and the organizers) might use other formats, if they fit their internal systems.

But if we take look at international solutions, like searching the most appropriate public transport connection in Google Maps, the most frequent one is GTFS - general transit feed specification, which was already mentioned in the introduction (section 1.4)

As it turns out, the main advantage (and difference) of GTFS is that it is able to specify the stop locations without relying on external data source, such as national database of stops, which does not exist in the Czech Republic yet. [1]

The TTP therefore has to keep its internal database of bus stops, which can be further edited by the user, with the advantage of being able to add currently non-existing stops.

In the future, it might be useful to implement the GTFS format as well without affecting the non-timetable parts of the TTP.

3.2 Timetable editors for organizing public transport

An example of very simple timetable editor which we were able to try out was BEZDĚZ [2], not available online anymore.

[1] https://openstreetmap.cz/talkcz/c3372/

[2] https://www.k-report.net/presmerovani/?prispevek=1886740

As it is mainly focused on train transport (while allowing to add bus lines as well), one of the interesting things is that the user inputs individual trips as already to blocks, so the bus scheduling is practically done before being able to print the timetables.

Since we have not been able to find and try out any simple enough timetable editor to generate data as JDF, we opted for already owned MS Excel to see create timetables in similar format they can be seen online.

Although considered a commercial software as well, the XLSX format is open and can be read by other software, including open-source ones, as mentioned in the introduction (section 1.4).

3.3 Public transport trip planning software

For users of public transport, there are two main use cases for processing of timetables. First of them is finding optimal connection between two points, that is series of trips (including walking) whose optimization can be classified in at least three ways:

- Arriving to the target destination as soon as possible (from current point in time)

- Minimizing the travelling time (if planning to use given connection in the future, or the passenger does not mind waiting)

- Minimizing the walking distance (for passengers with reduced mobility)

This does not include factors like barrier-free access, minimizing the time of transfers or even reliability of the connection, and the list of results might not be always fully satisfactory, so we can see this is another hard discipline and therefore not implemented in TTP, especially if we restricted ourselves to using only bus transport.

The other use case is displaying trips from a given stop, which tells the users how long they have to wait until the given bus (or other vehicle) comes. Such tools usually do not show the lines separately (as they would on timetables) they simply show the trips ordered by the departure time at a given stop. The amount of shown trips is usually limited by the panel size.

In the TTP, this use case is implemented in trip querying as to show all trips which are to come in the next 24 hours, from a stop selected by the user.

Such feature is already available on IDOS[3], which gives a fixed list of 20 trips incoming (and 10 more for each expansion of the list).

[3] https://idos.idnes.cz/en/vlakyautobusymhdvse/odjezdy/

In our TTP, we can consider this as an integration of the feature, while serving as an alternative test of correctness whether our input data are parsed correctly.

3.4 Optimization software

There are multiple instances of optimization software, such as Optibus[4], mentioned at pages 232–234 in [17], for vehicle and crew scheduling optimization.

A very recently updated and open-source software is Lintim[5], which allows integration of whole timetable organization from line planning to vehicle scheduling, as mentioned in [4].

[4] https://www.optibus.com/

[5] https://lintim.net/home

Chapter 4

Functional and non-functional requirements

Here we describe how are the functional and non-functional requirements of the application defined and met.

4.1 Functional Requirements

The functional requirements were defined directly in the thesis assignment.

4.1.1 Parsing of timetable in JDF

TTP shall be able to parse a timetable in JDF format and store it in memory in a structured way.

This function is necessary for many use cases (the following functional requirements).

The functionality is implemented in the `JDF_Conversion` package. It was tested only visually in debugger during the initial development, and then also by exporting the parsed data back to JDF format and comparing the files.

During the further usage of the application, the functionality was tested indirectly by the output provided by the other TTP modules.

A focus was given on the correct parsing of line and trip data which would be relevant for bus scheduling, and which are required in any kind of JDF batch.

The TTP should be also able to accept a valid input and reject an invalid one. In its current state, it determines the JDF validity based on the format of data:

- file names
- file format

- correct amount of columns
- correct field value type

but there are no guaranteed validations on relationship between data (such as non-descending stop times).

Since the validation is done by trying to parse and load the JDF data into memory, if the file passes upload validation, it is guaranteed to be validated in subsequent TTP usage, as the algorithm for parsing and validation is deterministic.

The parsing and validation is also guaranteed to work JDF 1.10 format as well, as some of the data on CIS JŘ were found still in this format. Fields present from JDF 1.11 which are missing in 1.10 are simply considered as empty, on the server the data are stored in already 1.11 version.

4.1.2 Handling of road network data and stops in OpenStreetMap

The TTP shall be able to load road network data from OSM with bus stops.

In TTP, the loading of road network and stops is done in separate steps. In order to know what part of road network is relevant for our problem, the TTP first needs to determine the area, which is in turn determined by the stop locations.

The automation of this procedure was developed during the schedule visualization, where the user was able to provide districts to search for stops in (using OSM API) so the stop locations were determined before the road network was loaded.

However, this search was still unreliable (not all stops might have the standard, or they might not even exist), so we kept the stop-searching part of the script only as an external script to handle most stops.

In the TTP itself, we assume the stop locations are input manually, with OSM only as a visual helper.

4.1.3 Trip and stop editing

The TTP shall be able to support editing trips and stops in the loaded timetable.

The user can add or remove stops from the internal database of stops in a separate module. There is no special support for editing stop names or locations, as adding a new one and removing the old one is sufficient (plus both actions can be done within one request).

For trip editing, the direct JDF manipulation via the interface is not supported. Neither is supported the editing of the trip data for scheduling (or importing them).

This feature is however partially supported by the export of the timetable to XLSX format, which can be then edited in a spreadsheet editor and re-imported back to TTP (as a new JDF batch, or overwriting the old one), allowing for a more user-friendly editing of the timetables than the direct JDF, although with some limitations.

4.1.4 Deadhead distance calculation between stops

The TTP shall be able to calculate the distance between two stops in the road network.

The functionality is directly implemented as a subpart of the scheduling calculation in two ways.

From the loaded JDF data, the TTP iterates through all trips and create an undirected graph network of stops, where each vertex corresponds to each stop, an edge exists whether there is a trip between two stops. The edge weight is a smallest departure/arrival difference between the two stops across all trips serving these stops.

The TTP then uses the Dijkstra algorithm to calculate the shortest path between all pairs of terminal stops.

The second way is to use the OpenStreetMap data to calculate the distance between two stops. We do not utilize loading the actual stop nodes in OSM, only the road network is enough.

From the stop locations supplied by the user, we can determine which area of the road network to download as a bounding box with extra kilometres added to the sides, to account for the fact that a path between two stops might lead outside the non-expanded bounding box.

Then we can calculate the distance between two stops as the shortest path in the road network, again using the Dijkstra algorithm, with the road speeds accustomed to the bus speeds (multiplied by the factor of 0.9 on slower roads and 0.8 on faster roads to account for worse bus acceleration).

4.1.5 Trip filtering

The TTP shall be able to filter trips in given time range or passing through given stops.

The filtering is directly accessible to the user (generally called Trip querying in the program) once the given JDF batch is loaded. The user can select the date range of trips to be shown. The result list of trips is then rendered by *datatable*[1] library, allowing for further filtering and sorting.

[1] https://datatables.net/

For filtering trips passing through given stops, the TTP supports departure or arrival from one selected stop within 24 hour interval.

4.1.6 Scheduling calculation

The TTP shall be able to calculate the lowest amount of buses needed to serve all trips within given time range. The scheduling shall also minimize the deadhead time and pull-out and pull-in times from single depot.

The scheduling prerequisites (calculation of deadhead times and trip filtering from time range) are related to the functionalities described in 4.1.4 and 4.1.5.

Both functions are used in the scheduling calculation, which uses algorithms described in chapter 5.

The user can then download the result as JSON and also see it directly in the browser, with the help of *datatables* to see how are trips distributed between the buses (=bus blocks).

4.1.7 Visualizing trips on map

The TTP shall be able to visualize the trips on an OSM background.

The visualization is done on the on a schedule-level, where one schedule file (in JSON) is sent to the TTP and the servers sends multiple files to the client (all in one zipped archive).

The OSM background requires internet connection, so two files (per bus block) are sent to the client: one contains an HTML file with renders colored lines connecting the trips on the OSM background, the other contains the output of `pyplot` plot saved as PDF which contains the identical lines without any background.

More detailed description of the visualization is in chapter 6.

The schedule-level processing is done as it supports the intended workflow of the TTP, to download the schedule and immediately upload it to the visualization module.

To visualize only one specific trip, the schedule file must be edited manually. The functionality to accept only trips directly would not be hard to implement, but would require more time and testing.

Generally this requirement is satisfied only minimally, as the schedule visualization has no labels (despite them existing in *matplotlib* library, they are not available in *mplleaflet*) or allowing to show/hide specific trips.

This was abandoned due to time constraints and high complexity of the task.

4.2 Non-Functional Requirements

We can also discuss some quality attributes of the application.

4.2.1 Page architecture

The TTP is a multi-page application, where each data exchange is done by a separate page load, so no AJAX requests are used.

Each module has a common prefix in the URL.

Whenever suitable, the request is created as a `GET` method, which has the advantage of showing query string in the URL, so skilled users can change the data directly in the browser instead of filling the form again.

This is also a main way to handle redirection in the application, which occurs after most form submissions.

The disadvantage can be that while `GET` signals idempotent request, it can sometimes take a long time to load the page due to the request complexity.

Whenever a file can be uploaded, the `POST` method is used.

4.2.2 Performance

The speed of the application is discussed in chapter 7. The main concern was for the scheduling calculation, which is the most demanding part of the application.

We can say that as the single-depot scheduling was completed in 40 seconds on the largest dataset, we can be satisfied with the result.

The most time-consuming part has sadly been the schedule visualization on map using OSM data, where downloading the road network data took the most time.

Regarding multiple requests at once, we discuss it in 4.2.5.

We should also warn against accidental clicks. Submitting two requests at once is not an issue for simple page loading, but if two scheduling requests are sent at once, there is no benefit in parallel processing,

4.2.3 Reliability

The TTP shall be reliable in the sense that it should not crash or produce incorrect results.

During the testing, the only suspicious results observed were related to scaling performance, where smaller dataset required more time to process than the larger one in one case (see section 7).

Handling of errors is done in multiple ways, sorted from user-friendliest:

- For expected errors to be thrown during the processing, the TTP catches them and displays them to the user with reasonable message.
- For other caught errors, the TTP catches them and displays them to the user with a Python-specific message.
- For uncaught errors, the application returns a 500 error.

Most of the crashes are prevented directly by the Flask framework, which catches them and returns a 500 error.

The user can easily restart the server in case of a crash or too many concurrent requests, following by refreshing the browser page.

4.2.4 Internet connection requirement

In the TTP, styling tools such as *datatables* are downloaded locally, so the application can run without internet connection, but require manual update in case the user wants to use a newer version of the library.

Once data are sent from the server to the client, the client can work offline.

An internet connection is however needed for any kind of OSM usage. The map visualization can be done without internet connection (drawing only direct lines), but to see the actual map on the rendered HTML, the user must be connected to the internet.

4.2.5 Scalability

Generally the application was initially meant to be single-user. Using the browser allows for some scalability, but the server-side processing is not optimized for multiple users.

We initially did not worry about that, as allowing multiple users would also be a matter of security (see 4.2.6).

The Flask framework is designed to support multitasking by using e.g. `celery`[2], so we assume a possible extension of the application to support multiple requests at once in a reasonable time.

4.2.6 Security

The application uses browser features for uploading and downloading files, so the server does no directly read or write data outside its directory without the user's consent.

[2] https://flask.palletsprojects.com/en/2.3.x/patterns/celery/

The names of schedules, trips and maps generated at the server are sanitized to prevent directory traversal attacks.

Accidental sending of nonsense data is prevented by the validation of the JDF format. A possible risk might occur during the JDF batch selection (e.g. for trip querying), where the user is offered a dropdown list of JDF batches. We must consider security in two ways: the security of the server data and security of the user data.

The server data are stored persistently on a filesystem, which can be directly accessed by the user. By being designed as a single-user application, we did not care about preventing this.

We should however consider the security of a normal application usage. We might use the scheduling calculation as an example. The first issue is concurrent writing to the files, in case of accidental double-request submission. Based on the operating system, the file might be locked during writing, so only one request would fully succeed and the other would immediately throw an error once the server tries to open the file.

We therefore rely on the operating system to handle the file locking.

Another issue might be a directory traversal attack, where the user would try to access files outside the designated directory by adding `../` to the file path. The place where such input could be is in all forms where name of a JDF batch is selected (including the upload form).

Finally, for a truly online application, managing of user accounts would be necessary to implement and to make a separate folder for each user.

4.2.7 Usability

The application uses standard HTML forms and `datatables` for the user interface, allowing for well-documented and user-friendly approach. No special optimizations were made for mobile application, as it is expected to run on a desktop.

4.2.8 Maintainability

The application is separated into 4 packages:[3]

1. `JDF_Conversion`
2. `Map_Visualization`
3. `Bus_Scheduling`

[3] by the history of development, from oldest to newest

4. `Browser_Interface`

A programmer documentation is available in the chapter 10.

On multiple places, type hints in Python are used to help when using IDE.

A bit untraditional is the usage of camel casing for most of the original code (author's personal style), Line breaks for longer expressions do not have any general style, it is a balance between readability and line length.

The application allows replacing most of the parts with new ones, as the interface of functions is well-defined.

Unfortunately, most changes require direct code editing, as the application does not support any kind of plugin system.

4.2.9 Interoperability

The application runs on Python 3.10, which is a common version of Python and available on most systems. The user interface is only a browser one. This theoretically allows for sending any HTTP request to the server, but the actual structure of the responses is not documented (no direct data access via API).

Chapter 5

Scheduling calculation

The process of scheduling is the computationally hardest and also the most interesting part of the program. We reiterate what was mentioned in the introduction, and how three models are implemented:

- Default scheduling (DS) the simplest model, where the only constraint is the feasibility of chaining trips.
- Single depot scheduling (SS) the model where each bus has to return to the same depot.
- Circular scheduling (CS) the model where the buses do not have to return to the depot, but can stay near the first stop of their first trip.

5.1 Statement of the problem

Given a list of trips, we are supposed to assign one bus to each trip. Using notation from Bunte and Kliewer [18], we denote following:

- $i, j:$ trip indices
- $u, v:$ stop indices
- s_i : start station of trip i
- e_i : end station of trip i
- d_i : departure (beginning) time of trip i
- a_i : arrival (final) time of trip i
- t_{uv}: travel time from station u to station v

These variables represent our input data - data provided to this section of TTP.

Furthermore, we use in the following sections:

- B - a set of bus blocks (collection of ordered collections of trips),
- n - amount of trips
- m - amount of buses

The most important relation, which can be derived from our input variables, is compatibility relation - denoted as $i\alpha j$ which is satisfied if trips i and j can be served after each other (chained). It can be easily seen that this means $a_i + t_{e_i s_j} \leq d_j$. Having the trips ordered by departure time, as we assume further, also implies that $i < j$ is a necessary condition.

This relation can be manually constrained further by the scheduler, if for practical reasons some trips should always or never be chained. In TTP, we however simply assume that all pairs of trips satisfying the aforementioned inequality can be chained.

This allows us to state a simple two-phase optimization problem. Since there exist multiple models which are equivalent - such as maximum matching being convertible to maximum flow algorithm, or minimum weight perfect matching to linear sum assignment problem, we will use a model that might not have 1:1 correspondence to any of the cited sources, but fits the implementation in TTP, which calls algorithms from the `NetworkX` library on bipartite graphs.

5.2 Minimizing amount of buses

We start by representing our set of trips as bipartite graph G, where one trip is represented by two vertices.

- There exists a set of depots D. For SDBS, $D = \{d\}$ as our only chosen depot. For MDBS, we would have $D = S$.
- Each trip i is represented by a pair of vertices x_i and y_i.
- For a pair of trips i, j, there exists an arc (x_i, y_j) iff $i \, \alpha \, j$.

We plan to represent schedules by making "arrows" (arcs) through the trips. A block for one bus is an ordered list of trips, therefore the arcs should represent disjoint paths. With no arc selected, we have achieved a trivial schedule - assign one bus to one trip. This is almost never optimal, so we want to reduce the amount of buses used. This means that we want to select maximum amount of arcs, where each vertex has maximally one outgoing and one incoming arc.

This selection yield us a subgraph, where amount of weakly connected components is equal to amount of vertices with no incoming arc. Each component is therefore an ordered path, representing a schedule of an individual bus.

(show example)

An i-th trip is represented by two vertices. The vertex x_i represents an arrival to e_i and the vertex y_i represents a departure from s_i.

Since each trip must be completed (served), we could assume an arc (y_i, x_i) would be always present. This can already represent a feasible solution, where each bus serves one trip.

Yet we aim for a solution where each bus serves as many trips as possible. A chaining of trips would therefore mean selecting as many arcs (x_i, y_j) as possible, then the solution could be easily read as a set of ordered paths, each representing a bus block.

By ignoring the naturally completed edges (y_i, x_i), we have created a bipartite graph G[1].

The amount of edges in the graph is asymptotically quadratic to the amount of trips. The upper bound is given by n arrival vertices and n departure vertices, which at the best case[2] can be matched each x_i with y_j if $i \leq j$.

Having the graph as bipartite allows to specify our objective as finding maximum cardinality matching (further shortened to maximum matching) of G. This can be easily calculated by Hopcroft-Karp algorithm in $O(n^2.5)$ time [19], which is implemented in Python library *networkx*. The matching will be denoted as $M(G)$.

An example of the scheduling graph can be shown on the figure 5.1. (green table is a list of trips, blue table is a list of deadhead times):

The solution from created matching can be read as:

5.3 Counting with deadheads

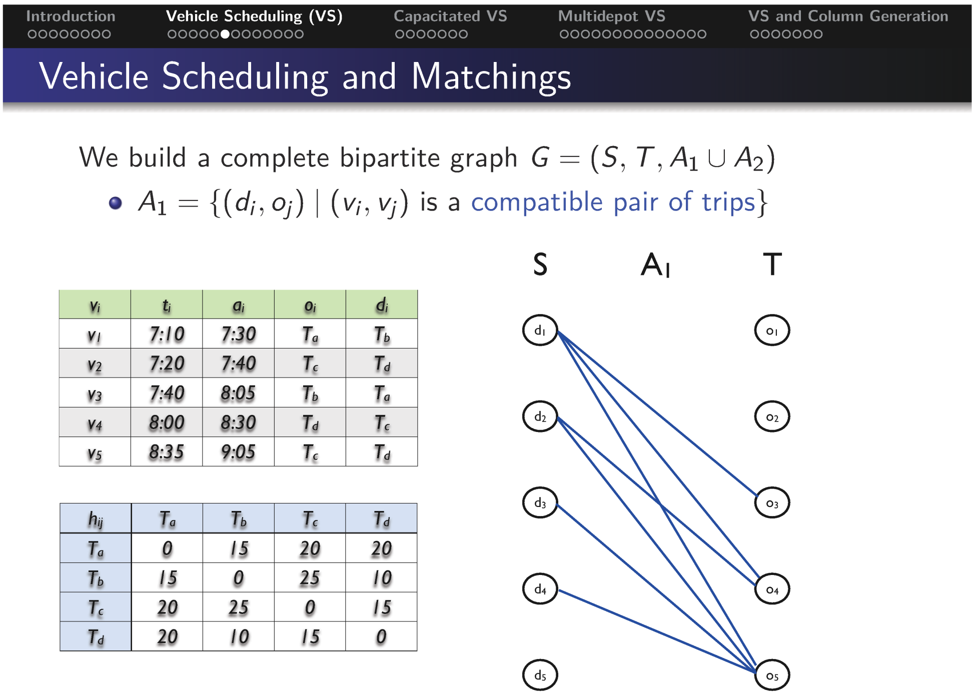
If the only requirement for chaining trips is the mentioned feasibility relation, we can consider the amount of minimizing bus amount as solved.

In Peřina [20], it is shown that more buses would be required to use if we wanted to avoid aggressive driver switching with respect to the mandatory breaks.

We focus on if our trip chaining is optional under other aspect: the deadhead time. This is a time needed for the bus to travel from a final stop of its trip to a first stop of its next trip. To minimize time spent driving between trips, the bus would

[1] Note the graph is always disconnected, y_1 and x_n have no incident arcs.

[2] Best case for potential optimization, worst case for complexity



Introduction Vehicle Scheduling (VS) Capacitated VS Multidepot VS VS and Column Generation

Vehicle Scheduling and Matchings

We build a complete bipartite graph $G = (S, T, A_1 \cup A_2)$

- $A_1 = \{(d_i, o_j) \mid (v_i, v_j)$ is a compatible pair of trips$\}$

v_i	t_i	a_i	o_i	d_i
v_1	7:10	7:30	T_a	T_b
v_2	7:20	7:40	T_c	T_d
v_3	7:40	8:05	T_b	T_a
v_4	8:00	8:30	T_d	T_c
v_5	8:35	9:05	T_c	T_d

h_{ij}	T_a	T_b	T_c	T_d
T_a	0	15	20	20
T_b	15	0	25	10
T_c	20	25	0	15
T_d	20	10	15	0

Figure 5.1 Example of scheduling graph. Source: Stefano Gualandi https://dm872.github.io/assets/dm872-vehicle-scheduling-handout.pdf

Algorithm 1 Reading the solution from the matching

```
$M \leftarrow$ matching from $G$
$B \leftarrow \emptyset$
$i \leftarrow 1$
while $i \leq n$ do
    $b \leftarrow \emptyset$
    while $x_i$ is not matched do
        $b \leftarrow b \cup \{i\}$
        $i \leftarrow i + 1$
    end while
    $B \leftarrow B \cup \{b\}$
end while
```

ideally end its trip at a stop where the next trip starts. Given our relaxed definition of stop (a point on road, not a given platform), we specify our turnaround time as zero ($t_{u,u} = 0$). In general, the deadhead time is considered a measure.

As mentioned in the basic example, all trips have to be served. This means that to reduce total driving time, we can only reduce it by reducing deadhead time. If we used our bipartite graph example and assigned deadhead time to arcs ($w(y_i, x_j) = t_{a_j d_i}$), we would be searching for minimum weight maximum cardinality matching. Using further helper vertices, we can convert this problem to minimum weight perfect matching, which is even easier to solve.

We define graph G' as:

- For each trip i, we have vertex x_i and y_i.

- For a pair of trips i, j there exists an arc (y_i, x_j) iff $i \, \alpha \, j$ with weight $w(y_i, x_j) = t_{a_j d_i}$.

- Given an amount of m required buses, we create vertices b_k for each $k \in \{1 \ldots m\}$.

- For each bus k and trip i,there exists an arc (b_k, x_i) and an arc (y_i, b_k), both with zero weight.

We can easily see this graph always has a perfect matching, this means $M(G') = n + m$: As we have calculated that $m = n - |M(G)|$, each pair of unmatched vertices in G is responsible for adding two more vertices into G', one in each partition. If we construct the matching in G' based on the matching of the original vertices in G, our extra vertices can be always matched with the originally unmatched ones.

Our implementation also allows to adjust the edge weight based on waiting time. In the actual experiments, a small waiting time coefficient (of 0.001 extra cost per minute of waiting) was added to each edge, to prioritize minimum downtime as a tertiary objective in case of equivalent solutions.

5.4 Adding pull-out and pull-in

So far have we assumed that the buses start their first trips anywhere. But usually, at the end of their shift they have to visit their depot (garage). This depot can be considered a regular stop in our distance function (deadhead matrix). For this, we can simply set our weights ín graph G' as $w(b_k, x_i) = w(y_i, b_k) = t_{Dx_i}$. We can be sure all depot vertices (representing pull-in and pull-out) be covered, as their amount is identical to the amount of uncovered vertices of the original solution, therefore allowing for a perfect matching.

5.5 Multiple depots

When introducing multiple depots, the conditions introduced to our problem make it hard to solve the problem quickly. Each bus would need to return to the same depot it started in, which is not a condition that is easily introduceable to our previous models. Various models can be found in literature, such as Bunte and Kliewer [18], to tackle this NP-hard problem (the NP hardness is proven in Bertossi, Carraresi, and Gallo [5]).

A solution (neither interface nor algorithm) for multiple arbitrary depots is not implemented in TTP . However, in the following section we show an alternative real-life problem whose solution could be used to solve the multiple depot problem.

5.6 No depots, circular scheduling

In regional transport, it might not be efficient to return the buses to the depots each day, especially if the region has a major central city (first morning trips lead to this city from the outer areas of the region while last evening trips lead out of this city). Therefore, we might assume the buses stay near the locations where first trip takes place. This might allow for efficient allocation of drivers, who can be hired from the respective outer towns and do not need to use individual transport vehicles (cars) to get to the place. Yet this requires a constraint that a vehicle block must have its final trip end at the place the first trip started, or at least incur a penalty equivalent to the deadhead time between last and first stop.

This formulation can be converted to multi-depot model, where each starting stop of each trip can be considered a depot. Since vehicle scheduling literature does not mention this specific case, we might assume no exact solution would be significantly faster. But due to its usefulness, this method of scheduling is kept in the TTP, using local search.

5.6.1 Local search for circular scheduling

As for initialization, we calculate the SDBS scheduling twice. First without a depot, then we pick a terminal stop which would be an optimal depot for the current solution, and finally recalculate the solution with the depot.

The local search then creates new solutions as following: We assume the bus can be scheduled to travel the whole area without returning to the starting stop, and that two buses might be scheduled to travel in the opposite direction. We therefore aim to find a solution to switch parts of their vehicle blocks so that their ending trips are exchanged, while also ensuring the swap is valid:

Let us have two vehicle blocks A and B with starting stops s_A and s_B, and ending stops e_A and e_B, respectively. The cost $c(A)$ is the sum of deadhead times during the whole block, so including the return times, the total cost for the whole solution is $c(A) + d(e_A, s_A) + c(B) + d(e_B, s_B)$ (where d is the return time). Each block can then be split into two parts, creating 4 vehicle blocks A_1 , A_2 , B_1 , B_2, which can be recombined as $A' = A_1 + B_2$ and $B' = B_1 + A_2$.

We can denote as last trip of A_1 to be t_i, first trip of A_2 to be t_j, last trip of B_1 to be t_k and first trip of B_2 to be t_l. For our swap to be valid, the feasibility relation must hold between $i \alpha l$ and $k \alpha j$, therefore the intervals $[i, j]$ and $[k, l]$ must overlap.

As j and l strictly depend on chosen i and k respectively, each swap can be defined by a pair of trips, yet we can assume the amount of available swaps will be significantly lower than amount of all edges, as seen on real-data in 7.6.

We have therefore changed two deadhead costs and two return costs. As the deadhead times were optimal, the initial swap will never improve them, but we can improve the return times.

Searching for a valid swap might be computationally hard and is done randomly, however if they are precalculated, all swaps related to non-swapped pairs of vehicle blocks can be carried to the next iteration.

We can also define the swap operation for cases where one of the four vehicle blocks is empty. The case analysis (impact on the total deadhead time) here is skipped for brevity, but is fully implemented in the code.

Having multiple empty vehicle blocks is not considered, as it would mean one of the buses would have completely empty blocks (which we have proven as impossible in our previous models), or the bus blocks would be completely switched.

Note that this approach would work also for unbounded multiple-depot scheduling, if instead of the return time we select the optimal depot, based on the first and last trip of the block (this can be precalculated).

5.6.2 Circular scheduling parameters

We use three simple parameters for the local search: Amount of iterations, number of new solutions branching from one solution, and amount of surviving solutions for the next iteration.

To determine the survivors, the best solution is always kept, the rest are chosen using tournament selection. The fitness is the same as in the default scheduling model - total deadhead times plus the return times.

5.7 Summary

The TTP supports the simplest cases of scheduling, where the only constraint on chaining trips is the feasibility of chaining trips. We are not comparing claiming one algorithm is better than another, as the constraints (or more precisely, expressions to be optimized) regarding pull-in and pull-out are different.

A simple thought experiment can be made, which effectively places the bounds on the circular scheduling.

5.7.1 Expected scheduling comparison results

By finding an optimal solution for single depot scheduling, we can use the identical bus blocks for both default scheduling and circular scheduling. For default scheduling, the pull-in and pull-out times are simply not considered for the deadhead times. For circular scheduling, the returning from last stop of the block to the first stop of the block is considered as a deadhead time, which will be more effective than pull-in and pull-out due to the triangular inequality. Finally, a default scheduling will have lower deadhead times than circular scheduling, as we can simply ignore the return times.

In the chapter 7, we will compare our scheduling efficiency on urban and regional transport datasets by default scheduling and single-depot scheduling, then show how much time can be saved by using circular scheduling, as compared to the single depot scheduling, and how incomparably low is the default scheduling as lower bound.

5.7.2 Time complexity

For the concept of single-depot scheduling, the time complexities are well known. The amount of trips is labeled as n, the amount of buses is labeled as m.

The bipartite graphs we create during the scheduling are dense (worst-case, each trip can be chained with all other trips that follow), so the amount of edges is $O(n^2)$.

The maximum weight matching to determine the amount of buses takes $O(n^{2.5})$ time with the Hopcroft-Karp algorithm. [3].

With added pull-out and pull-in edges and vertices, the minimum weight perfect matching can be calculated in $O((n + m)^3)$ time using Jonker-Volgenant algorithm. [4]

[3] used function is on https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.bipartite.matching.maximum_matching.html

[4] https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.bipartite.matching.minimum_weight_full_matching.html

These complexities also apply to the circular scheduling, with the added complexity of the local search.

For all solutions together, we keep $O(n^2)$ external memory to quickly calculate the returning edge weights for the circular scheduling, although we no longer use the graph for matching.

Each solution takes $O(n + m)$ memory to store the bus blocks, a constant amount for the fitness, and at most $O(n^2)$ memory for the swaps (lower in practice, as hinted in the subsection 5.6.1). Based on the distribution of trips between bus blocks, we could make an exact amount as such:

$|S(B)| = \sum_{i=1}^{m} \sum_{j=i+1}^{m} ((|B_i| - 1) \cdot (|B_j| - 1) + 2 * |B_j| + 2 * |B_i|)$

where $S(B)$ is the set of all swaps, B is the set of bus blocks, and $|B_i|$ is the amount of trips in block i. The final addition of $2 * |B_j| + 2 * |B_i|$ is to account for the possibility of one empty sub-block, either before the first trip or after the last trip. This is also why we subtract 1 from the amount of trips in the multiplication, as selecting the last trip would mean the rest of the block is empty.

If we assume the lookup time of deadhead times (or turn-around time) to be constant, the time complexity of calculating the fitness of a solution is constant as well.

For passing the solution to the next iteration, we need to update the set of viable swaps. This is done by excluding all trips that are part of the swapped blocks - this unfortunately is linear to the amount of previous swaps, although we no longer need to calculate the feasibility relation for each swap.

The addition of new swaps would require feasibility checks would only to the two swapped blocks, which is again based on the trip distribution.

Chapter 6

Map visualization

In this chapter, we show the implementation of map visualization and presented difficulties.

6.1 Introduction

With the context of public transport, we can find some usages of map visualization. First of them is visualization of most optimal route between two objects, which can be seen for example on figure 6.1.

Another use case is displaying a routing of a public transport line, as seen on the figure 6.2.

6.2 Our goal of visualization

In our TTP, we have decided to explore these use cases with a goal of visualizing bus schedules as trips on the map, which can be useful to introduce drivers to a new route, and to estimate where a bus would be at a given time.

We have decided for simple and readable solution, where we simply draw lines between stops and color them based on the time such trip is served. This way, we can easily see the progress of the bus on the route.

This was estimated to be quickly done with the help of `pyplot` module in `matplotlib` Python library, and these lines would then be projected on OpenStreetMap background using `mplleaflet` library.

Complications arise when a bus uses the same route multiple times in the schedule, as the lines would overlap. This is a rather common case, so we have chosen three solution so the amount of trips taken and the respective times can be still easily inferred from the visualization.

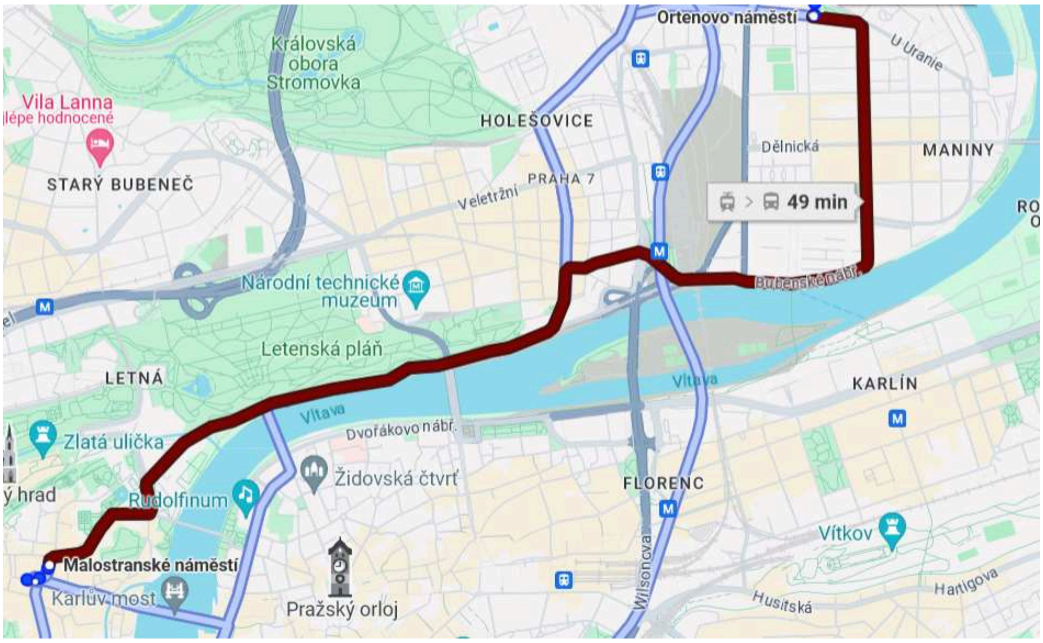


Figure 6.1 Google Maps route visualization of connection route. Source: Google Maps

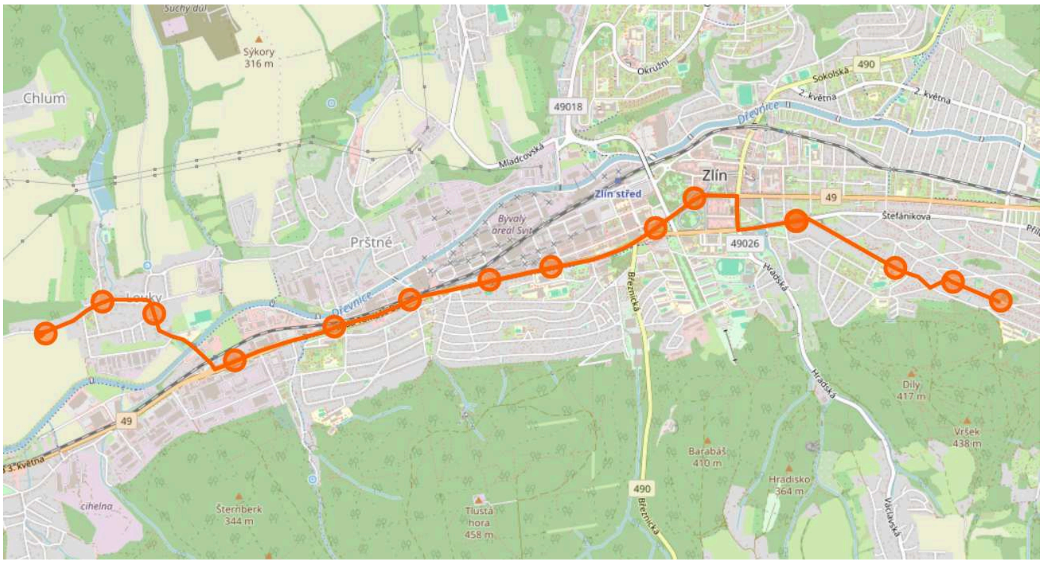


Figure 6.2 OpenStreetMap route visualization of a bus line. Source: OpenStreetMap

Issues can also arise with undefined stops. As the bus scheduling (distance matrix) has either a reasonable fallback on either using timetable data, plus we only need the location of terminals (whose count is less than total amount of stops), we could have simply rejected the request and asked the user to provide the missing data.

In the map visualization, the amount of missing stops might be higher, so we have decided to simply ignore the stops and treat them as if they were not there, to get at least default solution.

6.3 Data preparation

To keep it simple, we use the output from bus scheduling (see 5) as input for the map visualization.

Each bus block from the schedule is then treated separately (to be rendered on separate map data), and each trip from the bus block is treated as a sequence of trips between two stops. If two consequent trips need a deadhead, it is also automatically inserted.

6.4 Data rendering

We decided to render the trip as a line segment between two stops with these parameters:

- width based on the average speed of the trip (max 80 km/h, min 0 km/h)
- color based on the time when the trip is served (rainbow color map, red at 00:00)
- solid line for regular trip, dashed line for deadhead
- points: by default, starting and ending stop (with respective x and y parameters as latitude and longitude)

Unfortunately, `mplleaflet` does not support labels which could show the number of respective trips, so this was abandoned due to time constraints.

6.4.1 Overlapping routes

We have decided for 3 solutions on how to display the overlapping routes, each available for the user. As overlapping route, we simply define a trip which starts and ends at the same stops as another trip, or such trip goes in the opposite direction. As mentioned before, each trip is considered to only contain two stops.

1. The trips are condensed into line segment, which is marked with multiple colors (each color corresponds to different subtrip, same as the solutions below)
2. The trips are drawn as parallel - this has the disadvantage of the starting and ending location of the line segment not being directly on the stop location
3. overlapping routes between stops share the same start and end point, but are represented as curves (arc with different radii).

The options preserve the other parameters (line width, color, style) as described in the previous section, but change the location of first and start endpoint. Since the third option also requires curve rendering, it was simplified to be rendered as jagged line segments (with enough segments to make it look like a curve).

6.4.2 Navigation data

Instead of straight lines between stops, it can sometimes feel more natural to make the trips follow the actual road network. Given we already use OpenStreetMap for different use cases(shortest route calculation), it would also make sense to actually render this shortest route.

It was done by finding the nearest point of the road network to the stop, and then using the `networkx` library to find the shortest path between these points, and render the trips as passing through these points (as if they were stops). This can be combined with the rendering options for overlapping routes (see 6.4.1).

The result was not satisfactory, as the road network is not always accurate, and the shortest path can sometimes lead to a different road than the bus would actually take, but is kept in the code for possible future improvements.

As can be seen in chapter 7, the rendering of the trips can be quite slow, even if we download the whole map for all bus blocks at once.

Without exact testing, there could be three approaches to downloading the OSM data for the map visualization:

1. Download the whole map data for the whole schedule at once
2. Download the whole map data for each bus block separately
3. Download the whole map data for each trip separately

As our scheduling algorithm does not place any restrictions on how distant stops each bus can serve, the expected benefit of serving small areas would be outweighed by the need to redownload the data.

Chapter 7

Performance testing results

7.1 Introduction

We were checking the bus scheduling application performance with respect to some chosen bus transportation systems. The systems are chosen by real data as following:

- **Small town urban transport**: Uherské Hradiště (2023) (further as **U1**), 366 trips
- **Larger urban transport**: Havířov (2023) (further as **U2**), 1438 trips
- **Small district transport**: Uherské Hradiště (2023) (further as **R1**), 1880 trips
- **Large regional transport**: Plzeňský kraj (2020) (further as **R2**), 3495 trips

The data are taken from official CIS JŘ data.

The point of this thesis is not to design an optimization of the existing systems, therefore details are kept away. The reasoning for choosing **U1** is author's familiarity with the system (for easy testing purposes).

The **R2** was obtained as an older dataset for testing purposes from the transport agency Arriva in the Pilsen region.

As **U1**,**U2**, and **R1** have their timetables valid for the same year, the relevant tested processes will use the same dates in the further sections. For **R2**, equivalent dates were chosen (to maintain the same day-of-week status and school holidays).

7.2 Testing environment

For performance testing, a lab computer was used as as a server:

- Operating system: Linux 5.15.146 Gentoo
- Processor: Intel(R) Core(TM) i5-4570S CPU @ 2.90GHz
- RAM: 8 GB
- Cores: 4
- Python version: 3.10.8
- Flask version: 2.3.2

As a client, a laptop was used:

- Operating system: Windows 10 Pro
- Processor: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
- RAM: 8 GB
- Cores: 4
- Browser: MS Edge 124.0.2478.80 64-bit

The server calculations were measured as performance-time by the Python `time` module directly within the server code. The client time was measured using the browser extension `Page load time` [1] on MS Edge browser.

For shorter calculations (below 1 minute), the time was measured as an average of 5 runs.

7.3 Trip querying

The trip querying gives us an approximate view on the overall JDF batch data and the efficiency of the software.

To do this, we query all trips, arrivals, and departures both all at once and separately.

For accurate trip count, a January week (from Monday to Sunday) was queried, where fewest exceptions can be expected (school holidays, etc.).

The departures and arrivals were counted for 24 hours from Friday to Saturday, to cover both workday and weekend on average.

[1] Version 3.0.0 https://microsoftedge.microsoft.com/addons/detail/page-load-time/llcdjocbfkdndmjbgpaibfkdjkjogeho

Dataset	U1	U2
Trip records [a]	366	1438
Trips from	2023-01-09	2023-01-09
Trips to	2023-01-15	2023-01-15
Arrival stop	Uherské Hradiště„aut.nádr.	Havířov,Podlesí,aut.nádr.
Arrival date/time	2023-07-07 13:30	2023-07-07 13:30
Departure stop	Uherské Hradiště„Studentské nám.	Havířov,Město,Nemocnice
Departure date/time	2023-06-01 13:00	2020-01-17 13:00

Dataset	R1	R2
Trip records	1880	3495
Trips from	2023-01-09	2020-01-13
Trips to	2023-01-15	2020-01-19
Arrival stop	Uherské Hradiště„aut.nádr.	Plzeň„Terminál Hlavní nádr.
Arrival date/time	2023-07-07 13:30	2020-07-17 13:30
Departure stop	Kunovice„Na Rynku	Klatovy„Tylovo nábřeží
Departure date/time	2023-06-01 13:00	2020-01-17 13:00

Table 7.1 Querying data for trips, departures, and arrivals

[a] The amount of *Trip records* is the number of lines in the file `Spoje.txt` in the JDF data.

For more variance, different stop and date was chosen for departure and arrival. The departure stop is a frequented stop, while the arrival stop is always an important terminal. Since most trips arriving at the terminal end there, the arrival count is usually lower than on pass-through stops, if we assume equal frequency (as on pass-through stops, the departure/arrival would be considered twice, once in each direction).

The exact data are shown in Table 7.1.

First, we show the time needed to query and retrieve all trips, departures, and arrivals, to give an idea of the approximate size of the whole dataset, and to show the performance bottleneck.

We can see that even though trip count gets higher, the correlation between trip amount and server time does not correlate between **U2** and **R1**.

This might be due to **U2** having more weekend trips (which are usually less frequent in regional transport) which do get counted less times, but are iterated over more often.

On the other hand, the significantly higher client time is mostly dependent on the trip count, as the `datatables` library needs to render the data.

Further, we see how much faster we get when querying all data at once.

Obviously, querying arrivals and departures separately as well would lead to

Dataset	U1	U2	R1	R2
Departure+Arrival count	168	467	366	216
D+A server time	544	3 121	1 258	2 325
D+A total time	1 822	4 700	3 125	3 696
Trips count	1 063	4 916	7 159	14 265
Trip count (avg 1 day)	151	702	1 023	2 424
Trips server time	616	3 911	2 313	5 289
Trips total time	2 739	10 848	13 137	16 796

Table 7.2 Querying result of trips, departures and arrivals

Dataset	U1	U2	R1	R2
Trip count (1 week)	1 063	4 916	7 159	16 966
Trip count (avg 1 day)	151	702	1023	2424
Arrival count	87	192	241	95
Departure count	81	275	125	121
Server time (ms)	538	4 485	2 593	5 485
Server time separately	1 160	7 032	3 571	7 614
Total time (ms)	2 802	10 977	7 312	16 968
Total time separately	4 561	15 548	16 262	20 492

Table 7.3 Querying result of trips, departures, and arrivals

Dataset	U1	U2	R1	R2
Trip data	366	1 438	1 880	3 495
Terminal stops	11	38	120	405
TT time (ms)	26	354	254	674
OSM time (ms)	5 931	18 610	65 723	837 264

Table 7.4 Distance matrix calculation results

even higher total time.

This confirms that the actual overhead lies in the code loading the JDF data, a disadvantage that was to be expected when designing the application.

7.4 Distance calculation

The distance calculation is already directly relevant to the scheduling optimization.

In short, we calculated 2 types of distance matrix - either from timetable data further shortened as *TT matrix*, and from map data, further as *OSM matrix*, as mentioned in the functional requirements (see 4.1.4). Without surprises, calculating the TT matrices (directly using JDF data) is faster.

In the table 7.4 we show the time needed to calculate the distance matrix for each dataset. only the time spent directly on the map calculation is shown.

A slightly unfair thing in the comparison is that we have removed the overhead for unpacking the JDF data, whereas OSM would only need to load the list of trips and stop locations. but that would not reduce the whole overhead caused by the need to download the OSM data.

The average and maximum distances are shown in the figure 7.1.

Regarding the distance matrix calculation based on timetable data, we again see an anomaly between **U2** and **R1**, that the regional dataset is calculated faster than the urban one.

For matrix calculation, the distance does not matter on the actual trip data, but on how large the target area is, which is easily illustrated again on the **U2** and **R1** datasets - despite the similar trip count, the largest distance between two terminals is almost 4 times larger in **R1**, which also leads to a larger calculation time. [2]

This raises a question if the distance calculation via OSM was worth it. Good news is that the distance matrix is cached on the server and can be also manually

[2] Since the longest travel time (using the best path) would be probably between two points near opposite corners, the amount of nodes and edges in the road network would scale quadratically with the travel time.

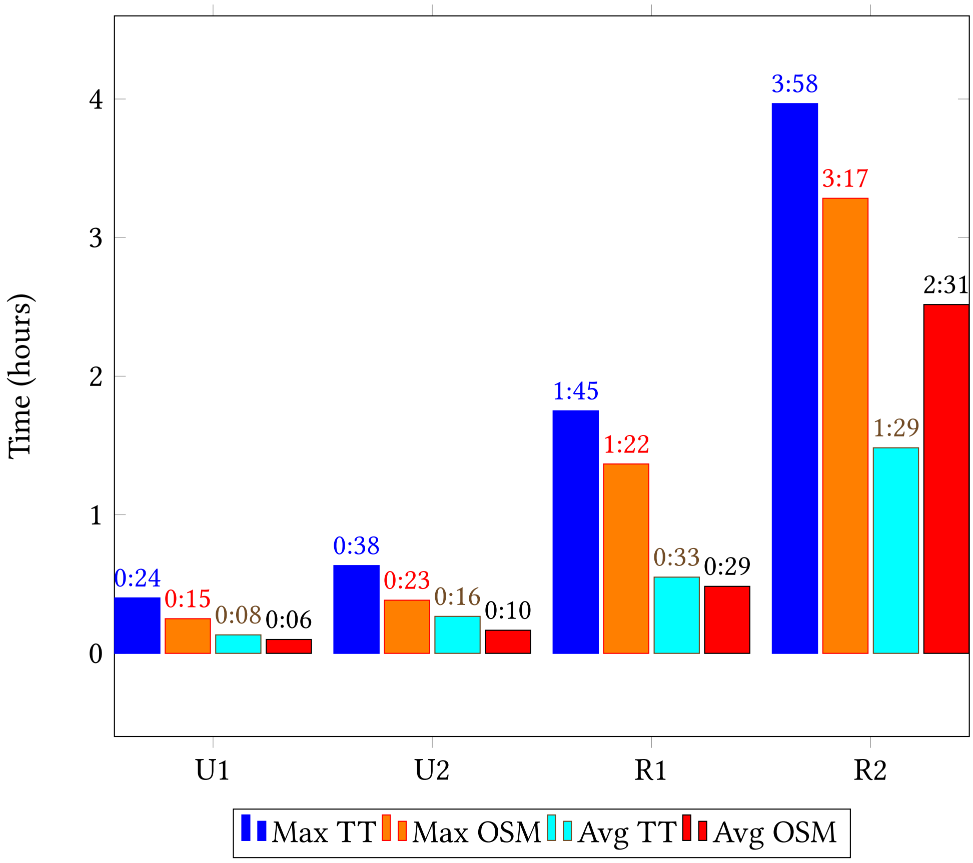


Figure 7.1 Comparison of Maximum and Average Distances in Timetable and OSM

input, so this was not necessary to do every time new schedule calculation was made.

As suggested in the introduction (1.2.1), the buses might not always take the shortest route, so the distance matrix actually leads to better result than using the timetable in the sense of the actual distance between stops being lower. This is confirmed in our table, as each dataset has lower maximum distance between two terminals when calculated via OSM .

It should be noted that for the **R2** dataset, the stops *Letiny,,u kostela[PJ]*, *Plasy[PJ]*, and *Skašov[PJ]* were disconnected, so the distance between either of the stops and any other was undefined (=very high value). In the results, their respective distance for the TT matrix was manually replaced by the OSM matrix distance.

The average distance via OSM is also lower for the first three datasets. Despite being higher for the **R2** dataset, the OSM calculation will be shown to be more effective with regards to the actual scheduling.

7.5 Single depot scheduling

For a single depot scheduling, we have chosen a single day of trips and to properly exclude night hours. The table 7.5 shows the overall trip count in each dataset and their total duration.

The chosen depots are:

- **U1**: Uherské Hradiště,,garáže ČSAD
- **U2**: Havířov,Podlesí,aut.nádr.
- **R1**: Uherské Hradiště,,garáže ČSAD
- **R2**: Plzeň,,Slovany

For **U2**, the arrivals are cut off at 00:30 to avoid continuous night trips (for other data, the default 24 hours was enough).

Following the dataset introduction, in the table 7.6 we show the actual time taken by the scheduling calculation (only OSM matrix):

As we estimated in chapter 5.7.2, the amount of chainable pairs scales indeed quadratically with the number of trips (figure 7.2), and the time needed for calculation scales cubically (figure 7.3).

This however does not tell us how useful were the additional amount of edges we gained by using the OSM matrix.

Regarding the client performance, the rendering overhead is not as significant as in the trip querying, since the datatable only renders the table of bus blocks, while the trips are displayed only upon expansion.

Dataset	U1	U2	R1	R2
Input day	2023-09-01	2023-09-01	2023-09-01	2020-01-13
Input time range	00:00-00:00	04:00-00:30	00:00-00:00	01:00-01:00
Actual first departure	04:48	04:00	03:43	04:43
Actual last arrival	22:55	00:14	23:24	00:10
Trip count	191	801	1294	2701
Total trip time (h:mm)	64:15	380:13	617:45	1372:18
Avg trip time (h:mm)	0:20	0:28	0:29	0:30

Table 7.5 Scheduling input data

Dataset	U1	U2	R1	R2
Trip count (nodes)	191	801	1294	2701
Chainable pairs (edges)	17 413	299 612	748 902	2 869 854
Bus count	10	41	91	204
Scheduling time (ms) - no depot	239	2 652	10 928	34 858
Scheduling time (ms) - one depot	236	2 907	11 649	36 822
Server time (ms) - no depot	781	3 590	11 996	36 414
Server time (ms) - one depot	620	3 299	12 888	37 950
Total time (ms) - no depot	2 040	4 981	14 785	38 900
Total time (ms) - one depot	1 868	4 920	14 983	39 842

Table 7.6 Scheduling calculation time

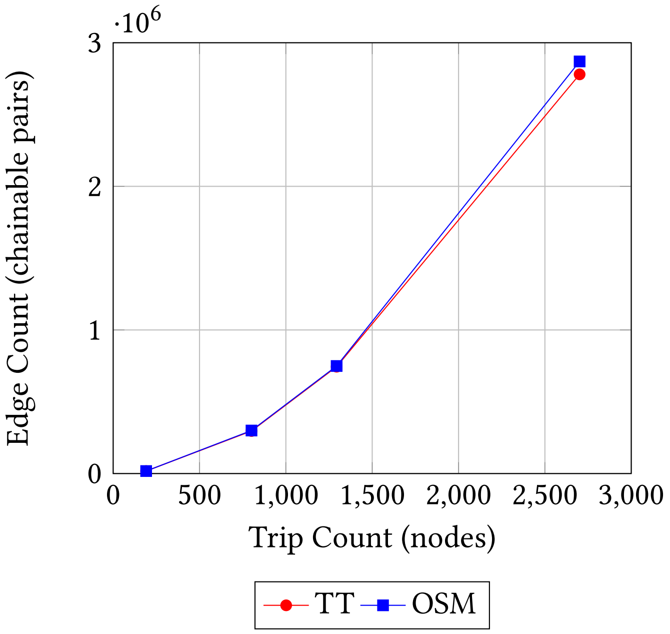


Figure 7.2 Edge count based on trip count

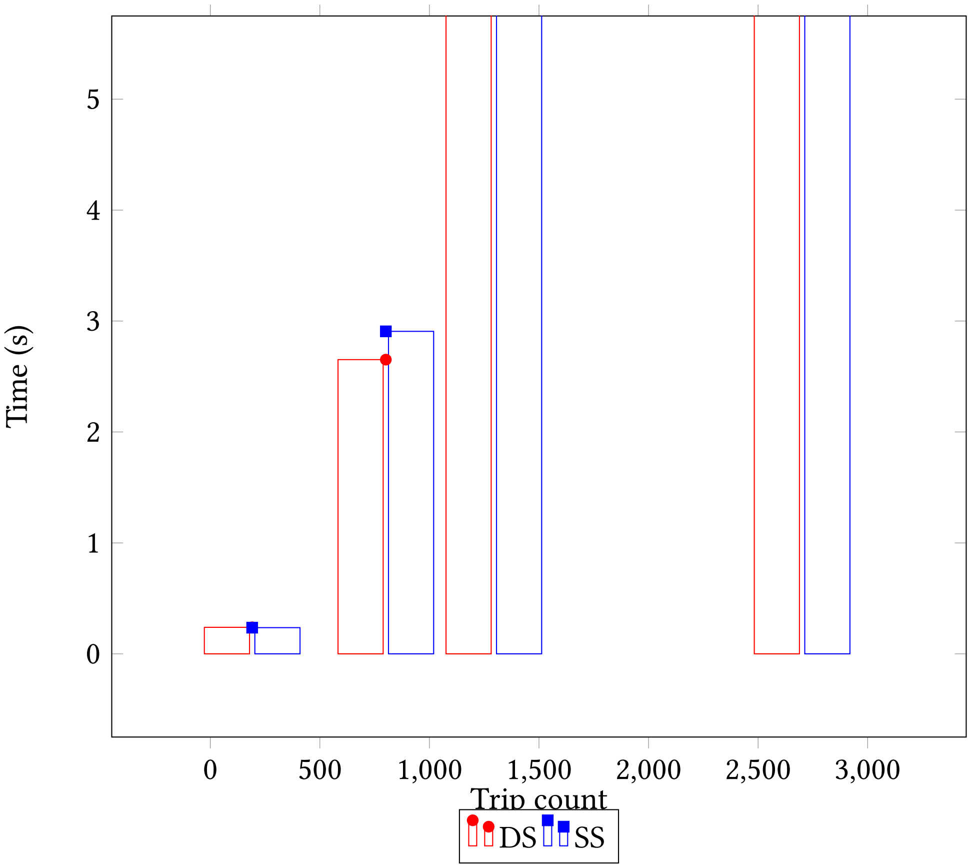


Figure 7.3 Scheduling calculation time based on trip count

Dataset	U1	U2	R1	R2
Trip count (nodes)	191	801	1294	2701
Chainable pairs (edges) - TT	17 356	296 391	744 132	2 779 709
Chainable pairs (edges) - OSM	17 413	299 612	748 902	2 869 854
Total trip time (h:mm)	64:15	380:13	617:45	1372:18
Bus count - TT/OSM	10	41	91	206/204
Deadhead time - TT	0:06	1:42	2:01	20:52
Deadhead time - OSM	0:04	1:11	1:03	17:25
Waiting time- TT	48:29	155:17	564:08	1409:05
Waiting time - OSM	48:31	165:39	567:27	1384:33
Productivity - TT	57 %	71 %	52 %	49 %
Productivity - OSM	57 %	70 %	52 %	50 %
Minimum driving time - TT	1:36	0:33	0:58	1:39
Minimum driving time - OSM	1:38	0:33	0:57	3:01
Maximum driving time - TT	12:42	15:02	13:30	15:33
Maximum driving time - OSM	12:45	15:02	13:31	15:17
Average driving time - TT	6:26	9:18	6:48	6:45
Average driving time - OSM	6:25	9:18	6:48	6:48

Table 7.7 Scheduling efficiency without a depot

7.5.1 Scheduling efficiency

We now present the overall scheduling results, which were calculated both using the OSM and TT distance matrices.

First, let us show the results of scheduling without a depot, on the figure 7.7. Avoiding pull-in and pull-out cost gives us lowest bound on the scheduling as a whole - if we consider driving times as accurate, the scheduling is constrained only by the laws of physics.

The difference between optimization using the TT and OSM matrix is clear. The **U1** did not offer much deadhead savings, as the time was very low in the first place, but all the other datasets reduced the deadhead time significantly. In the case of **R2**, even less buses could be used.

Solving the scheduling problem also tells us the overall productivity of the system, which is here calculated as the time spent serving trips compared to the total time (we do not count the waiting time between end of a bus block and the start in the next day).

We can think of the productivity as being relevant term, as we minimized the amount of buses needed - otherwise we could reach 100% productivity by simply adding more buses.

But since the waiting time is rather an issue of the driver than the bus itself (the

Dataset	U1	U2	R1	R2
Chainable pairs (edges) - TT	17 356	296 391	744 132	2 779 709
Chainable pairs (edges) - OSM	17 413	299 612	748 902	2 869 854
Bus count - TT/OSM	10	41	91	206/204
Deadhead time - TT	0:54	10:12	66:31	497:27
Deadhead time - OSM	1:11	6:35	65:04	384:07
Waiting time - TT	48:29	139:06	497:52	1339:41
Waiting time - OSM	48:31	156:41	549:56	1374:05
Productivity - TT	57 %	72 %	52 %	43 %
Productivity - OSM	56 %	70 %	50 %	44 %
Minimum driving time - TT	1:36	0:33	0:58	1:39
Minimum driving time - OSM	1:38	0:33	0:57	3:01
Maximum driving time - TT	12:42	15:02	13:30	15:33
Maximum driving time - OSM	12:45	15:02	13:31	15:17
Average driving time - TT	6:30	9:31	7:31	9:04
Average driving time - OSM	6:32	9:26	7:30	8:36

Table 7.8 Scheduling efficiency with single depot

driver still needs to be paid), we can set a reasonable upper bound on productivity as 8-hour shift with 30 minutes mandatory break after 4 hours of driving (as in a Czech safety break law [21]), which would make the upper bound 94 %.

Moving on to scheduling with depot, the table 7.8 shows the results, where we can see that the pull-out and pull-in time will add significant deadhead time to the scheduling.

On the other hand, the waiting time is not significantly affected by the depot existence, as can be seen in the figure 7.4.

Unfortunately, the application does not directly show how much does the existence of the depot affect the total deadhead times, to show how much deadheads is done during the day to reduce the pull-out and pull-in time.

We preferred to make the scheduling methods to have common output format, so also the user interface would require some change. To calculate the result, subtract the pull-in and pull-out time from the total deadhead time per each bus.

7.6 Circular scheduling with local search

For the circular scheduling, we use the same data as in the previous section.

We have kept the random seed, multiplications and survivors constant, only thing changing is the amount of iterations (generations).

The results can be seen in the table 7.9 and the relative improvement from

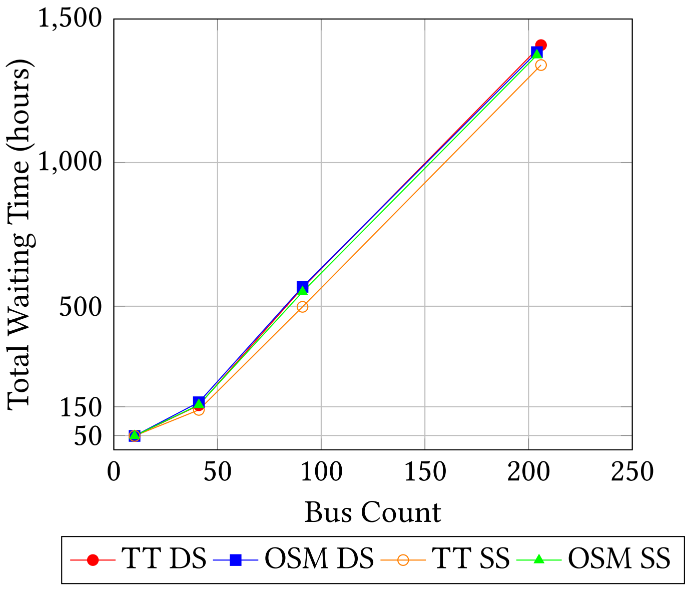


Figure 7.4 Waiting time based on bus count

the first iteration in the figure 7.5.

For more explanation:

- the amount of multiplications is 5
- the amount of survivors is 30
- the random seed is 1
- `Available swaps` is the amount of possible swaps in the initial solution, each iteration changes the solution by 1 swap at most.
- `Time per iteration` is the time needed to calculate the whole generation (was measured on 300 iterations).
- `Single depot` and `Default (no depot)` refer to the results calculated before (as upper and lower bound).

As we can see, the deadhead time still keeps improving with more iterations, at an insignificant expense of waiting time.

As mentioned in 5.6.1, the initialized solution is a single-depot scheduling with probably best starting depot (out of all terminals), so we have here the single depot scheduling (calculated in 7.8) as a reference.

We can also see the number of possible swaps is rather low (even lower than amount of buses multiplied by the amount of trips), compared to the amount of possible trip chains.

Dataset	U1	U2	R1	R2
Buses	10	41	91	204
Available swaps	967	11666	41951	90521
Time per iteration (ms)	133	842	1506	3465
Deadhead time (h:mm)				
Single depot	1:11	6:35	65:04	384:07
1 iteration	0:16	6:04	51:51	193:00
100 its.	0:09	4:57	39:21	177:43
200 its.	0:09	4:25	35:28	171:21
300 its.	0:09	4:20	33:05	161:29
Default (no depot)	0:04	1:11	1:03	17:25
Waiting time (h:mm)				
Single depot	48:31	156:41	549:56	1374:05
1 iteration	43:36	156:41	508:00	1371:29
100 its.	44:12	212:29	645:36	1382:52
200 its.	44:11	223:12	625:05	1396:37
300 its.	44:06	221:06	613:35	1398:49
Default (no depot)	48:31	156:41	549:56	1374:05

Table 7.9 Improvement of deadhead times by using circular scheduling

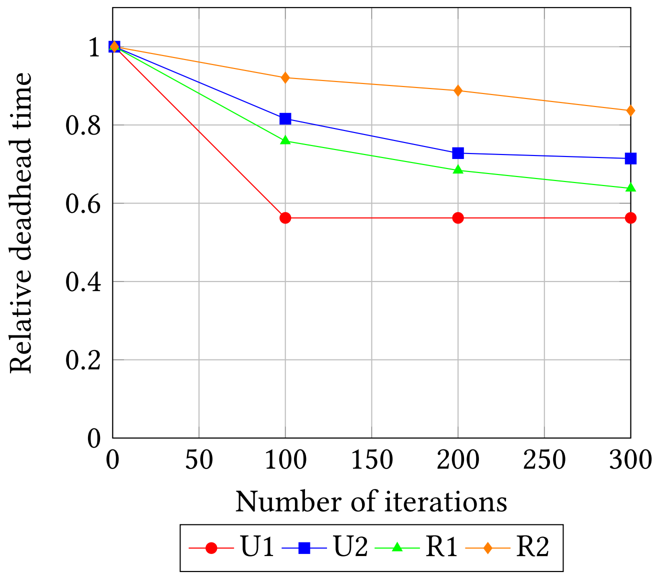


Figure 7.5 Local search improvement on circular scheduling

Dataset	U1	U2	R1	R2
Buses	10	41	91	204
Overlap - total time (ms)	5 203	29 764	42 331	78 075
Parallel - total time (ms)	5 193	28 646	41 917	78 479
Curves - total time (ms)	5 654	30 753	43 773	81 412
Overlap - size (KiB)	513	3 523	4 147	7 187
Parallel - size (KiB)	563	3 838	4 440	7 659
Curves - size (KiB)	2 155	15 103	14 051	23 152

Table 7.10 Time taken for rendering schedule on maps using direct lines, and size of the output file

7.7 Visualization

For visualization, we simply measured the amount of time taken to complete the whole request of rendering the map for the whole request, excluding the download, and also the size of the output file.

We can see the time taken for rendering is rather linear in the amount of buses, as each bus block is rendered separately, so the bigger distance of regional transport does not seem to play a big role here.

The increased size of the curves is due to the fact that the curves are not rendered as a single line, but as a series of small lines, giving the curvature illusion, as a chosen way of bypassing the limitation of the `mplleaflet` library.

This is in contrast to the time needed for rendering, as it is also higher than two other methods, but only by a small margin (10 % at most).

The next table shows the general trouble with using OSM navigation data for rendering, as something which theoretically works but is rather slow, similar to the distance matrix calculation.

We can see that the generic approach of downloading the whole map into memory and then finding shortest paths through our stop points is rather slow, although still viable for smaller maps (below 10 minutes for large urban dataset).

7.8 Summary

We can see that the TTP behaves consistently on different datasets and provides meaningful results, even though the time complexity is rather high.

As expected, the trip querying would be the fastest, followed by the scheduling and finally OSM integration. The application is not intended to be speed-critical, so the time taken, where reading the actual data by human takes longer than the calculation, is acceptable.

Dataset	U1	U2	R1	R2
Buses	10	41	91	3 [a]
Overlap - total time (ms)	32 222	442 982	3 831 049	557 714
Parallel - total time (ms)	32 330	413 116	-	-
Curves - total time (ms)	34 353	432 907	-	-
Overlap - size (KiB)	2 676	21 407	41 568	2 972
Parallel - size (KiB)	2 912	23 340	-	-
Curves - size (KiB)	12 444	99 586	-	-

Table 7.11 Time taken for rendering schedule on maps using OSM road network, and size of the output file

[a] Downloading the map took the most time, for 204 buses the time would not be 68 times higher.

Chapter 8

User documentation

TTP supports most actions to be done independently. These are:

1. Upload of timetable in JDF
2. Merging of JDF timetables
3. Export of timetable as XLSX
4. Upload of timetable in XLSX format
5. Stop editing
6. Scheduling calculation
7. Visualizing trips and schedules on map

Here are shown the main features of the application, which are available via the user interface. For the installation and launching instructions, see the chapter9. The directory and file structure of the data saved by the server application is described at the end of this chapter (section 8.10). More advanced users can edit the data directly.

8.1 Main page

The main page (at **Home**) is the starting point of the application. It contains a small introduction of the application. Every module can be reached from the navigation at the top.

8.2 General notes

The data shown on in tables (on trip querying and schedules) use DataTables library, which allows sorting, filtering, and pagination. For this, JavaScript must be enabled in the browser. It is also required for the correct processing of the stop editing and visualization.

8.3 Upload of timetable in JDF

This module is reachable on the navigation by the title **Upload JDF**. Te main purpose is to offer a way to upload JDF timetables on the server.

As was mentioned in the introduction, they are officially available at ftp:ftp.cisjr.cz For extensive searching, the website hrefwww.portal.radekpapez.czwww.portal.radekpapez.cz is recommended, as it allows to search for specific lines in order to be downloaded as JDF and also previewing of the timetables.

The JDF batches are uploaded as folders. Clicking on the **Upload** button opens a file dialog, where the user can select a folder. For more folders, click the + button.

They can also be uploaded as nested (pack more JDF batches into one folder and submit the parent folder).

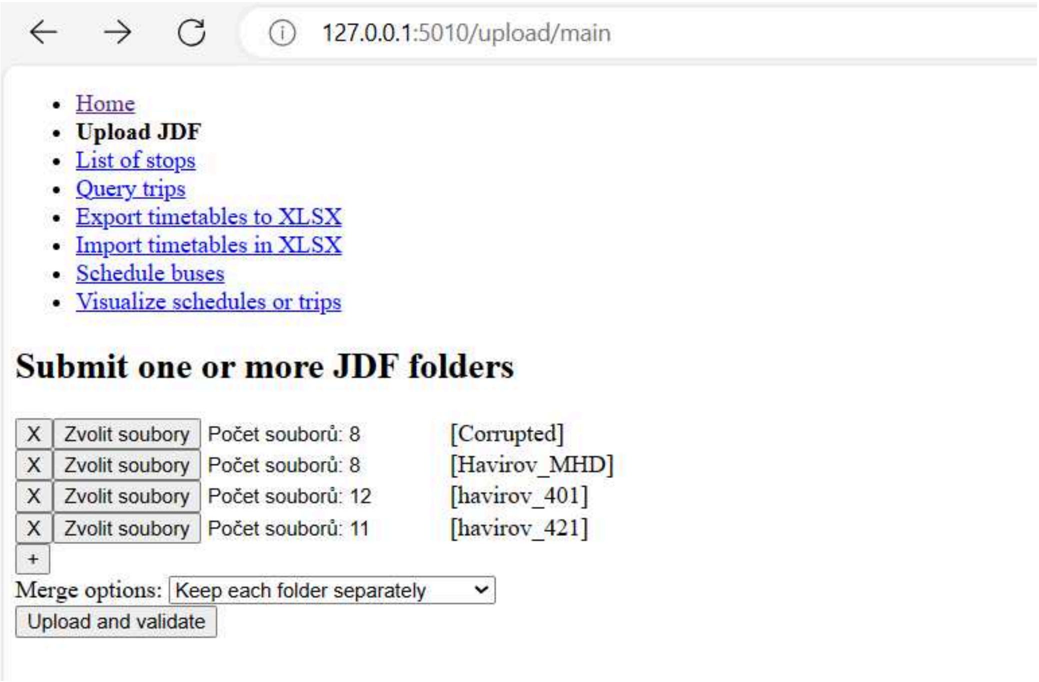
The figure 8.1 shows the form for uploading JDF batches.

Once the form is submitted, each folder is checked independently and the message of success or failure is displayed. The TTP does not warn when uploading a batch with the same name as an existing one, it is simply overwritten - the user gets informed about this in the final message. An example result in shown in the figure 8.2. Trying to upload two identically named folders will show a client-side warning.

In order to create a merged JDF batch (which would contain all trips from all uploaded batches), select a suitable option from the dropdown menu **Merge options** and fill up the name of the new batch. The option *Merge the contents into one JDF* will merge all uploaded batches into one and not keep the original ones on the server. In order to preserve both, select the option *Merge and keep.* For a successful merge, all uploaded batches must be valid.

The uploaded batches can be then used in modules:

- Export of timetable as XLSX (8.7)
- Trip querying (8.4)
- Scheduling calculation (8.5)



127.0.0.1:5010/upload/main

- Home
- **Upload JDF**
- List of stops
- Query trips
- Export timetables to XLSX
- Import timetables in XLSX
- Schedule buses
- Visualize schedules or trips

Submit one or more JDF folders

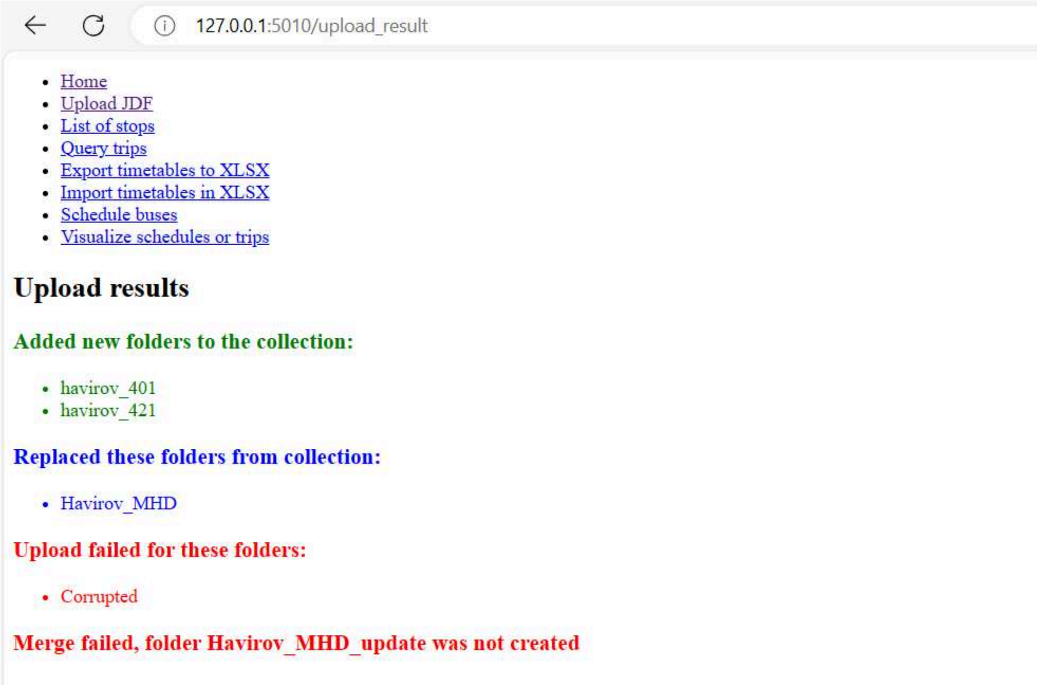
X	Zvolit soubory	Počet souborů: 8	[Corrupted]
X	Zvolit soubory	Počet souborů: 8	[Havirov_MHD]
X	Zvolit soubory	Počet souborů: 12	[havirov_401]
X	Zvolit soubory	Počet souborů: 11	[havirov_421]

+

Merge options: Keep each folder separately

Upload and validate

Figure 8.1 Form for uploading JDF batches. The folder "Corrupted" is supposed to represent invalid JDF.



127.0.0.1:5010/upload_result

- Home
- Upload JDF
- List of stops
- Query trips
- Export timetables to XLSX
- Import timetables in XLSX
- Schedule buses
- Visualize schedules or trips

Upload results

Added new folders to the collection:

- havirov_401
- havirov_421

Replaced these folders from collection:

- Havirov_MHD

Upload failed for these folders:

- Corrupted

Merge failed, folder Havirov_MHD_update was not created

Figure 8.2 Results of uploading JDF batches

8.4 Trip querying

This module is reachable on the navigation by the title **Query trips**. It allows to show all trips inside one of the uploaded JDF batch to the user within specified date range. The form on the page contains three sections. First is selection of JDF (the dropdown is based on the uploaded JDF batches), which needs no further explanation.

Second part allows for global trip querying. To confirm you want to use this function, tick the box *Show all trips* and then fill the range From - To with the desired values. Note the date range is inclusive.

The third part allows for an independent querying from the second part. The user can query trips which serve a given stop during a 24-hour time window.

The name of the stop does not need to match exactly, closest result is selected after submission. However, if such stop is already present in the 8.6 module, it will show an autocomplete help (list of stops matching the part name). This feature uses the `datalist` HTML element, which might not be supported in all browsers.

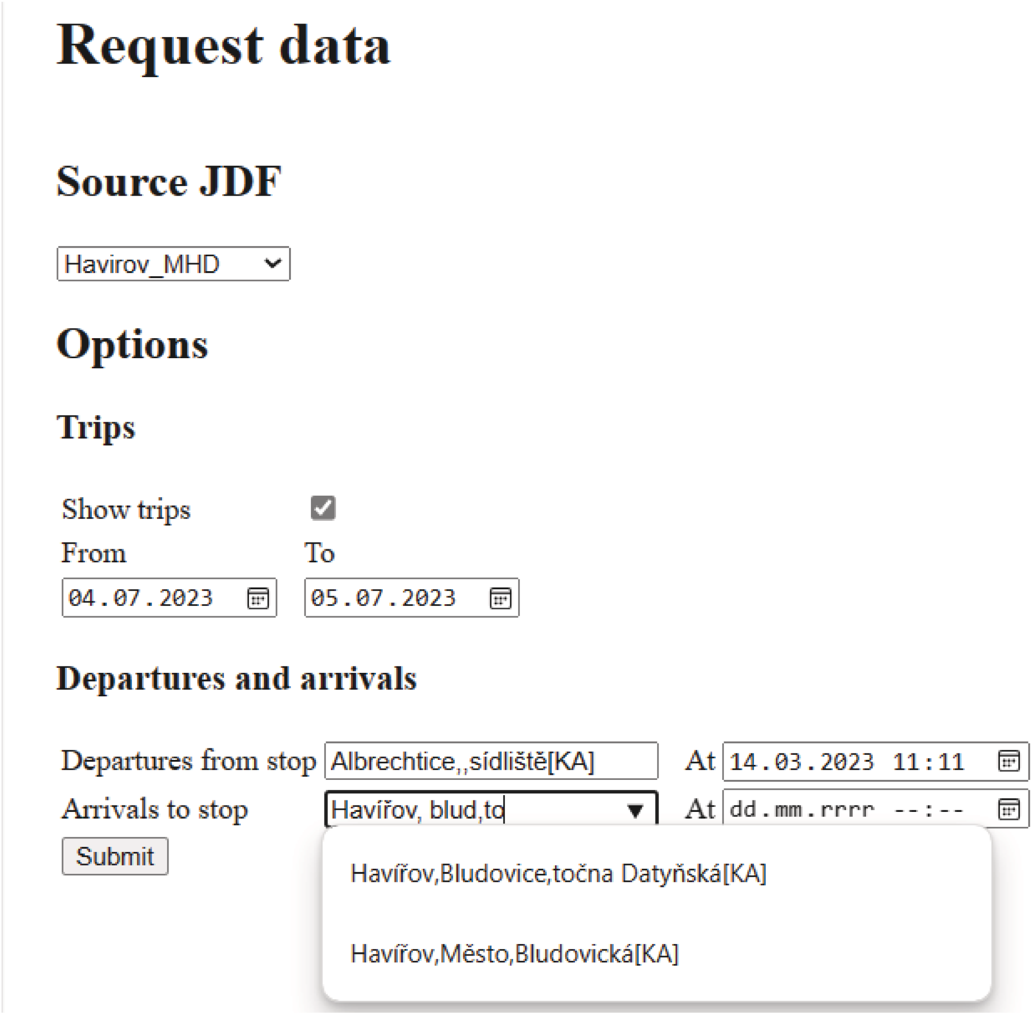
The date and time are in the format *YYYY-MM-DD HH:MM:SS*.

In the figure 8.3 we can see the user is about to submit a query on 2 days of all trips in the JDF batch *Havirov_MHD* and also all trips which departing from the stop *Albrechtice„Sídliště*. The arrivals part is being currently filled in (notice the datalist).

Submitting the form redirects the user to the page with the results. The page layout is always identical regardless of which things were actually queried (all trips, departures, arrivals).

Each of the results has separate tab, as seen on the figure 8.4. The example currently shows the arrivals on a stop, whose name can be seen in the header. The data (columns are) as following:

1. Day of the operation
2. Time of arrival (or departure) on the stop supplied by the user
3. Line number
4. Trip number
5. Departure time (of the first stop of a given trip)
6. Initial stop of the trip
7. Arrival time (of the last stop of a given trip)
8. Final stop of the trip



Request data

Source JDF

Havirov_MHD

Options

Trips

Show trips ☑

From 04.07.2023

To 05.07.2023

Departures and arrivals

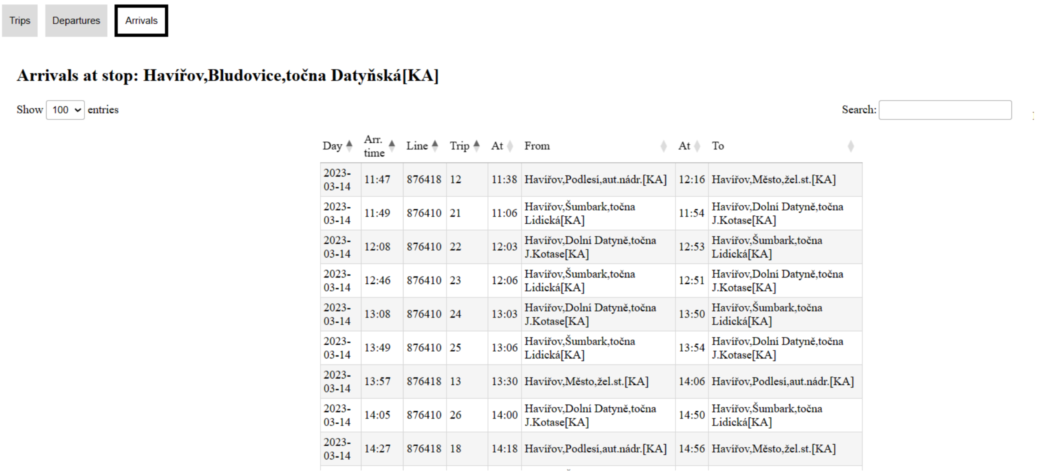
Departures from stop Albrechtice,,sídliště[KA] At 14.03.2023 11:11

Arrivals to stop Havířov, blud,to At dd.mm.rrrr --:--

Havířov,Bludovice,točna Datyňská[KA]

Havířov,Město,Bludovická[KA]

Figure 8.3 Form for querying trips being filled in



Arrivals at stop: Havířov,Bludovice,točna Datyňská[KA]

Show 100 entries

Search:

Day	Arr. time	Line	Trip	At	From	At	To
2023-03-14	11:47	876418	12	11:38	Havířov,Podlesí,aut.nádr.[KA]	12:16	Havířov,Město,žel.st.[KA]
2023-03-14	11:49	876410	21	11:06	Havířov,Šumbark,točna Lidická[KA]	11:54	Havířov,Dolní Datyně,točna J.Kotase[KA]
2023-03-14	12:08	876410	22	12:03	Havířov,Dolní Datyně,točna J.Kotase[KA]	12:53	Havířov,Šumbark,točna Lidická[KA]
2023-03-14	12:46	876410	23	12:06	Havířov,Šumbark,točna Lidická[KA]	12:51	Havířov,Dolní Datyně,točna J.Kotase[KA]
2023-03-14	13:08	876410	24	13:03	Havířov,Dolní Datyně,točna J.Kotase[KA]	13:50	Havířov,Šumbark,točna Lidická[KA]
2023-03-14	13:49	876410	25	13:06	Havířov,Šumbark,točna Lidická[KA]	13:54	Havířov,Dolní Datyně,točna J.Kotase[KA]
2023-03-14	13:57	876418	13	13:30	Havířov,Město,žel.st.[KA]	14:06	Havířov,Podlesí,aut.nádr.[KA]
2023-03-14	14:05	876410	26	14:00	Havířov,Dolní Datyně,točna J.Kotase[KA]	14:50	Havířov,Šumbark,točna Lidická[KA]
2023-03-14	14:27	876418	18	14:18	Havířov,Podlesí,aut.nádr.[KA]	14:56	Havířov,Město,žel.st.[KA]

Figure 8.4 Table with arrivals on a given stop

The table allows sorting, filtering, pagination, and to view the amount of results (at the bottom of page).

If the user fills in the stop name, but not date, an error will be shown (see figure 8.5), without affecting the other results:

8.4.1 Errors

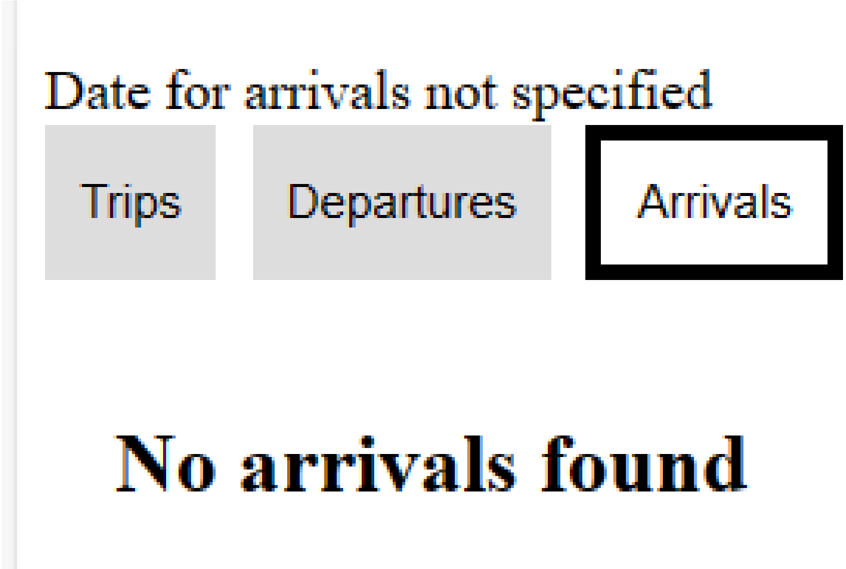
Common errors are:

1. Wrong date range: Timetables in JDF have limited date range (usually one year), so the query must be within this range.

2. No trips found even if date range is given: Do not forget to tick the checkbox *Show trips* if you want to see the trips.

3. `Could not find stop for a name <name>`:
   The stop name was not found in the JDF batch. The accuracy of the stop name is rather tolerant. Make sure you have selected the correct JDF batch.

4. `Could not find stop for a name <name from datalist>`:
   The stop name was not found in the JDF batch. As mentioned before, the datalist is constructed from the stops in the 8.6 module. The JDF batch might simply not contain the stop.



Date for arrivals not specified

No arrivals found

Figure 8.5 Error message when date is missing but arrival stop is filled in

8.5 Scheduling calculation

The scheduling calculation is available in the navigation under the title **Schedule buses**. The only pre-requisite for this is to have the correct JDF batch uploaded, which can be selected for the dropdown menu.

8.5.1 Date and time

The user can also select the date and time range for which the scheduling should be calculated. The maximum range is 1 day to ensure easy calculation and to prevent the user from waiting too long. For example, selecting the starting time as 2024-01-01 02:00 and the ending time as 01:00 will include trips, whose starting time (departure from the first stop) is after 02:00 of 1st January 2024, and their ending time (arrival in the last stop) before 01:00 of 2nd January 2024.

8.5.2 Deadhead matrix

The section *Method of calculating travel time between terminals* allows to specify how should the deadhead times be calculated, which the scheduling algorithm depends on. The method *Use timetable data* is the default one and requires no external tools, since everything is calculated from the saved JDF, as if bus speed between two stops was based on the quickest trip found in the timetable data.

Unlike shortest route algorithm used for searching passenger routes, the algorithm assumes the distance between two stops is based on the fastest trip between them (across the whole timetable data), and then further optimizes the distance between the terminals using Dijkstra's algorithm.

If the data need to be supplied more accurately, the method *Use uploaded matrix* can be selected. The format of the matrix is 2D CSV (actually tab separated values) with both row and column headers being a stop name, and the values being the travel time in minutes. The easiest way is to simply have the matrix generated by using the option *Use timetable data*, then edit it and resubmit.

Finally, if the timetable data might be too inaccurate (e.g. no route uses shortcuts, the route network is too radial) and manual work too tedious, an option is to select *Use OSM navigation* which tries to calculate the distance based on real road network data. For this, the actual stop locations need to be supplied (see 8.6). Be aware that retrieving the actual network for the first time (with disabled caching) might be too slow.

8.5.3 Scheduling goal

The section **Scheduling method** allows to select the method and goal of scheduling. The common objective for all is to minimize the bus and deadhead count (with respect to the queried trips). The difference is to reduce the pull-in and pull-out times:

- *Default,exact* - pull-in and pull-out not considered, always a consistent solution.

- *Depot,exact* - select a depot all buses have to start at and return to (times for these get included towards the total deadhead time). The depot is treated as a regular stop, therefore it must be included in the distance matrix (based on the method used in 8.5.2).

- *Circular,approximate* - each bus is scheduled to return to the place where it started the first trip of its block. The returning time is counted towards the total deadhead time. The solution is only approximate and uses local search, its parameters can be customized:

  - *Number of iterations* - determines the depth of the search, the higher the better solution, but also slower.
  - *Number of multiplications of each solution* - broadens the search space. One multiplication is one improvement - partially swapping two bus blocks. More to be seen in 5.

  - *Number of solutions kept* - how many solutions are stored in-memory per iteration. The best one is always kept, the others are randomly selected.

The search always starts with the default solution.

8.5.4 Invalidate cache

The server caches one distance matrix per JDF and a method. In order to force re-calculation (e.g. the JDF was overwritten), the checkbox *Invalidate cached distance matrix* can be ticked. This is not necessary if custom matrix is supplied, or different trips (terminals) are used.

The checkbox *Invalidate cached schedule* will also force the server to re-calculate the schedule. A schedule is cached by the parameters of the scheduling calculation (JDF, date range, method, goal), so it will be recalculated on its own often, a common use case could be however the approximate (heuristic,random) scheduling with identical parameters.

8.5.5 Precalculation

Before the actual scheduling is made, the server queries the necessary trips, calculates the deadhead matrix and a feasibility matrix (which trips can be served consequently by one bus). These are then stored in the cache and displayed to the user:

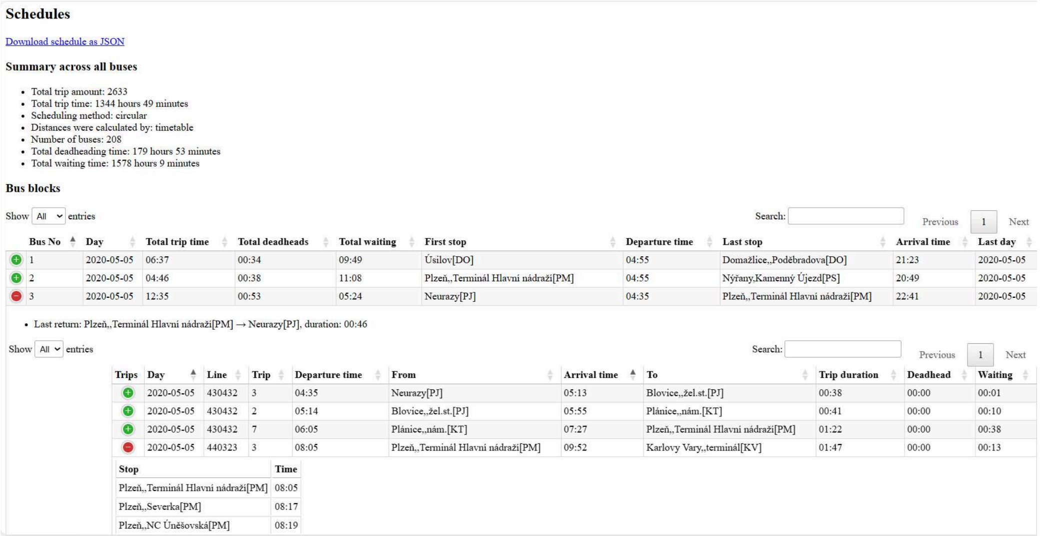
- Trips - the amount of trips in the selected date range

- Edges - the amount of edges in the deadhead matrix (affects the total scheduling time)

- First trip - the departure and arrival time of the first trip in the selected date range. Note that the departure time is referred to in the scheduling parameters, not the value input by user. E.g. if there is no trip between 00:15 and 00:30, then this value will be identical regardless of which time between 00:15 and 00:30 is selected.

- Last trip - the departure and arrival time of the last trip in the selected date range. The same note applies for the arrival time.

- Deadhead matrix - a download link to the calculated deadhead matrix. The downloaded matrix can be then directly uploaded in the next scheduling calculation, see 8.5.2.



Schedules

Download schedule as JSON

Summary across all buses

- Total trip amount: 2633
- Total trip time: 1344 hours 49 minutes
- Scheduling method: circular
- Distances were calculated by: timetable
- Number of buses: 208
- Total deadheading time: 179 hours 53 minutes
- Total waiting time: 1578 hours 9 minutes

Bus blocks

	Bus No	Day	Total trip time	Total deadheads	Total waiting	First stop	Departure time	Last stop	Arrival time	Last day
⊕	1	2020-05-05	06:37	00:34	09:49	Úsilov[DO]	04:55	Domažlice,,Poděbradova[DO]	21:23	2020-05-05
⊕	2	2020-05-05	04:46	00:38	11:08	Plzeň,,Terminál Hlavní nádraží[PM]	04:55	Nýřany,Kamenný Újezd[PS]	20:49	2020-05-05
⊖	3	2020-05-05	12:35	00:53	05:24	Neurazy[PJ]	04:35	Plzeň,,Terminál Hlavní nádraží[PM]	22:41	2020-05-05

- Last return: Plzeň,,Terminál Hlavní nádraží[PM] → Neurazy[PJ], duration: 00:46

Trips	Day	Line	Trip	Departure time	From	Arrival time	To	Trip duration	Deadhead	Waiting
⊕	2020-05-05	430432	3	04:35	Neurazy[PJ]	05:13	Blovice,,žel.st.[PJ]	00:38	00:00	00:01
⊕	2020-05-05	430432	2	05:14	Blovice,,žel.st.[PJ]	05:55	Plánice,,nám.[KT]	00:41	00:00	00:10
⊕	2020-05-05	430432	7	06:05	Plánice,,nám.[KT]	07:27	Plzeň,,Terminál Hlavní nádraží[PM]	01:22	00:00	00:38
⊖	2020-05-05	440323	3	08:05	Plzeň,,Terminál Hlavní nádraží[PM]	09:52	Karlovy Vary,,terminál[KV]	01:47	00:00	00:13

Stop	Time
Plzeň,,Terminál Hlavní nádraží[PM]	08:05
Plzeň,,Severka[PM]	08:17
Plzeň,,NC Úněšovská[PM]	08:19

Figure 8.6 Schedule result page

- Scheduling method - the method of scheduling with additional parameters.

To continue with the scheduling, simply click the *Submit* button.

8.5.6 Schedule results

The schedule result page contains three sections: download link, summary, and schedule table.

In the summary on the figure 8.6, we can see the "facts" as input data (number of trips and total driving time), then the "task" (how to schedule, and what was used for calculating the deadhead times), and finally the results (amount of buses, and total deadhead time).

The total waiting time is a time taken between two trips the bus is expected to not be driven.

The total deadhead time will differ based on the assignment. In case of scheduling with a depot, the pull-in and pull-out times are included, and in the case of circular scheduling, the returning times are included.

All times are displayed in the format hh:mm, while in the JSON data they are given as minutes only.

The schedule table is a nested table, which shows schedules for each bus. The outer table shows:

1. Bus number - the buses are ordered by their trip count, so the first bus is the one with the most trips.

2. Day of the first trip

3. Total driving/deadhead/waiting time - using the same definitions as in the summary

4. First trip - the departure time and first stop of the first trip

5. Last trip - the arrival time and last stop of the last trip

6. Day of the last trip - useful to prevent confusion when the last trip is on the next day (or multiday scheduling in general)

When expanding the bus block, there can be seen pull-in and pull-out times (if applicable), otherwise return time is shown. These times are included into the deadhead times as summary, unless using only the default scheduling without depot.

Under these, there is nested table with trips for given bus block.

Each trip is displayed with the same data as in the query results in 8.4 with three additional columns:

1. Trip time

2. Deadhead time (for reaching the first stop of the *next* trip)

3. Waiting time (for the *next* trip)

The trips in last row will always have the deadhead and waiting time 0, as there is no next trip.

Finally, each trip can be also expanded to show the list of stop it serves on the way (with the departure times [1]).

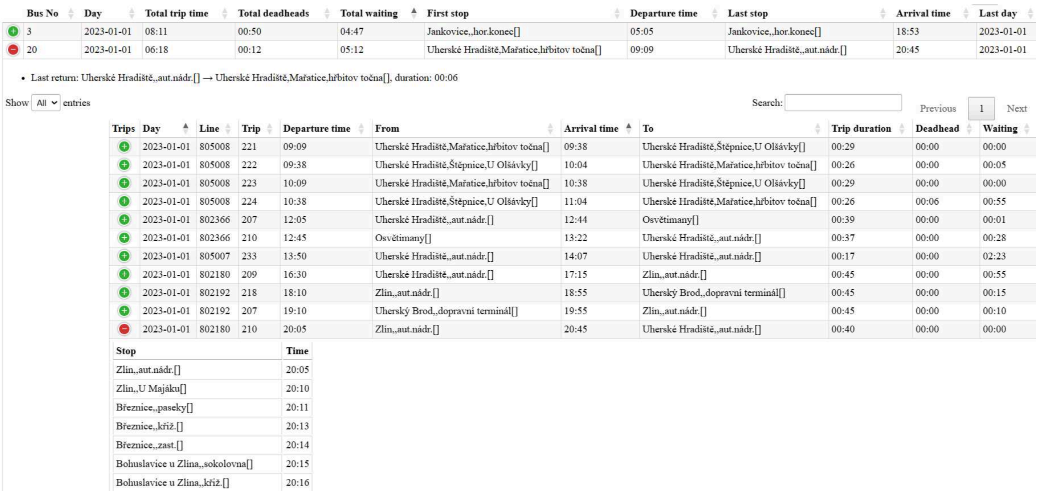
8.6 Stop editing

The stop editing is available in the navigation under the title **List of stops**. It is not necessary for basic JDF and scheduling function, however it is required for rendering schedules on map (see 8.9) and to calculate deadhead distance matrix using OSM data (see 8.5.2).

Note that a single stop in the TTP is assumed to have unified location (no specific platforms).

The whole page consists of 4 parts: the stop list, the stop editing map, the stop editing form, and the stop import form. The individual parts are easily seen from the image 8.8, the following documentation will rather focus on how they work together.

[1] Arrival time for the last stop



	Bus No	Day	Total trip time	Total deadheads	Total waiting	First stop	Departure time	Last stop	Arrival time	Last day
⊕	3	2023-01-01	08:11	00:50	04:47	Jankovice,,hor.konec[]	05:05	Jankovice,,hor.konec[]	18:53	2023-01-01
⊖	20	2023-01-01	06:18	00:12	05:12	Uherské Hradiště,Mařatice,hřbitov točna[]	09:09	Uherské Hradiště,,aut.nádr.[]	20:45	2023-01-01

- Last return: Uherské Hradiště,,aut.nádr.[] → Uherské Hradiště,Mařatice,hřbitov točna[], duration: 00:06

Show All entries

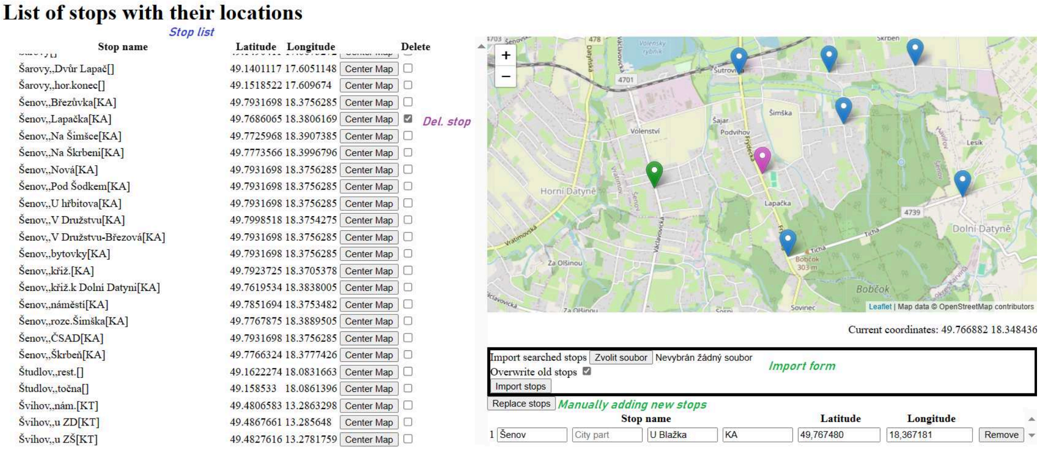
Search:

Previous 1 Next

Trips	Day	Line	Trip	Departure time	From	Arrival time	To	Trip duration	Deadhead	Waiting
⊕	2023-01-01	805008	221	09:09	Uherské Hradiště,Mařatice,hřbitov točna[]	09:38	Uherské Hradiště,Štěpnice,U Olšávky[]	00:29	00:00	00:00
⊕	2023-01-01	805008	222	09:38	Uherské Hradiště,Štěpnice,U Olšávky[]	10:04	Uherské Hradiště,Mařatice,hřbitov točna[]	00:26	00:00	00:05
⊕	2023-01-01	805008	223	10:09	Uherské Hradiště,Mařatice,hřbitov točna[]	10:38	Uherské Hradiště,Štěpnice,U Olšávky[]	00:29	00:00	00:00
⊕	2023-01-01	805008	224	10:38	Uherské Hradiště,Štěpnice,U Olšávky[]	11:04	Uherské Hradiště,Mařatice,hřbitov točna[]	00:26	00:06	00:55
⊕	2023-01-01	802366	207	12:05	Uherské Hradiště,,aut.nádr.[]	12:44	Osvětimany[]	00:39	00:00	00:01
⊕	2023-01-01	802366	210	12:45	Osvětimany[]	13:22	Uherské Hradiště,,aut.nádr.[]	00:37	00:00	00:28
⊕	2023-01-01	805007	233	13:50	Uherské Hradiště,,aut.nádr.[]	14:07	Uherské Hradiště,,aut.nádr.[]	00:17	00:00	02:23
⊕	2023-01-01	802180	209	16:30	Uherské Hradiště,,aut.nádr.[]	17:15	Zlín,,aut.nádr.[]	00:45	00:00	00:55
⊕	2023-01-01	802192	218	18:10	Zlín,,aut.nádr.[]	18:55	Uherský Brod,,dopravní terminál[]	00:45	00:00	00:15
⊕	2023-01-01	802192	207	19:10	Uherský Brod,,dopravní terminál[]	19:55	Zlín,,aut.nádr.[]	00:45	00:00	00:10
⊖	2023-01-01	802180	210	20:05	Zlín,,aut.nádr.[]	20:45	Uherské Hradiště,,aut.nádr.[]	00:40	00:00	00:00

Stop	Time
Zlín,,aut.nádr.[]	20:05
Zlín,,U Majáku[]	20:10
Březnice,,paseky[]	20:11
Březnice,,křiž.[]	20:13
Březnice,,zast.[]	20:14
Bohuslavice u Zlína,,sokolovna[]	20:15
Bohuslavice u Zlína,,křiž.[]	20:16

Figure 8.7 Expanded bus block and one trip on a schedule result



List of stops with their locations

Stop list

Stop name	Latitude	Longitude		Delete
Šarovy,,Dvůr Lapač[]	49.1401117	17.6051148	Center Map	☐
Šarovy,,hor.konec[]	49.1518522	17.609674	Center Map	☐
Šenov,,Březůvka[KA]	49.7931698	18.3756285	Center Map	☐
Šenov,,Lapačka[KA]	49.7686065	18.3806169	Center Map	☑
Šenov,,Na Šimšce[KA]	49.7725968	18.3907385	Center Map	☐
Šenov,,Na Škrbeni[KA]	49.7773566	18.3996796	Center Map	☐
Šenov,,Nová[KA]	49.7931698	18.3756285	Center Map	☐
Šenov,,Pod Šodkem[KA]	49.7931698	18.3756285	Center Map	☐
Šenov,,U hřbitova[KA]	49.7931698	18.3756285	Center Map	☐
Šenov,,V Družstvu[KA]	49.7998518	18.3754275	Center Map	☐
Šenov,,V Družstvu-Březová[KA]	49.7931698	18.3756285	Center Map	☐
Šenov,,bytovky[KA]	49.7931698	18.3756285	Center Map	☐
Šenov,,křiž.[KA]	49.7923725	18.3705378	Center Map	☐
Šenov,,křiž.k Dolní Datyni[KA]	49.7619534	18.3838005	Center Map	☐
Šenov,,náměstí[KA]	49.7851694	18.3753482	Center Map	☐
Šenov,,rozc.Šimška[KA]	49.7767875	18.3889505	Center Map	☐
Šenov,,ČSAD[KA]	49.7931698	18.3756285	Center Map	☐
Šenov,,Škrbeň[KA]	49.7766324	18.3777426	Center Map	☐
Študlov,,rest.[]	49.1622274	18.0831663	Center Map	☐
Študlov,,točna[]	49.158533	18.0861396	Center Map	☐
Švihov,,nám.[KT]	49.4806583	13.2863298	Center Map	☐
Švihov,,u ZD[KT]	49.4867661	13.285648	Center Map	☐
Švihov,,u ZŠ[KT]	49.4827616	13.2781759	Center Map	☐

Del. stop

Current coordinates: 49.766882 18.348436

Import searched stops [Zvolit soubor] Nevybrán žádný soubor
Overwrite old stops ☑
[Import stops]

Import form

[Replace stops] Manually adding new stops

	Stop name				Latitude	Longitude	
1	Šenov	City part	U Blažka	KA	49,767480	18,367181	Remove

Figure 8.8 Stop editing page

8.6.1 Stop map

The stop editing map connects to the OpenStreetMap API and displays the stops locations uploaded on the TTP server (the very same stops which are visible in the list on the left side).

Stops, which are currently saved on the server have blue markers. If a stop is set for deletion (by ticking the checkbox in the list), its marker turns purple.

You can zoom in and out, move the map, and hover on the markers to see the stop name. Clicking directly on the stop marker will center the map on it and also scroll to the stop in the list. Note: if the scrolling does not work, scroll with the right scrollbar. In order to find a stop not in the current view, use the button *Center map* in the stop list.

Clicking on an empty space will create a new stop (green). To discard it, click on the *Remove* button in the stop editing form.

8.6.2 Adding and removing stops

Clicking on an empty space in the map creates a green stop marker and also adds a new row to the stop editing form with pre-filled coordinates. These can be further modified in case of a missclick. In order to correctly submit the stop on the server, fill out its name (at least the city name). A comma will be interpreted as a separator, so the whole name can be without issue written just in the *City name* field.

In order to remove already uploaded stops, simply tick the *Delete* checkbox in the stop list. The stop will appear as purple on the map.

The button *Replace stops* will then send the request to server to both add and remove the stops. Direct editing of stop location or name is not possible, simply remove the stop and add it again.

8.6.3 Importing stops

The Import stops area is used for importing stops from the external script of stop finding (see A.3). The checkbox *Overwrite old stops* will remove stops with the same name as the imported ones.

8.7 Export of timetable as XLSX

The export of JDF is available in the navigation under the title **Export XLSX**. Exporting of the JDF allows to see the timetables for individual lines in human-readable way. For a default behavior, simply select the JDF batch name from the

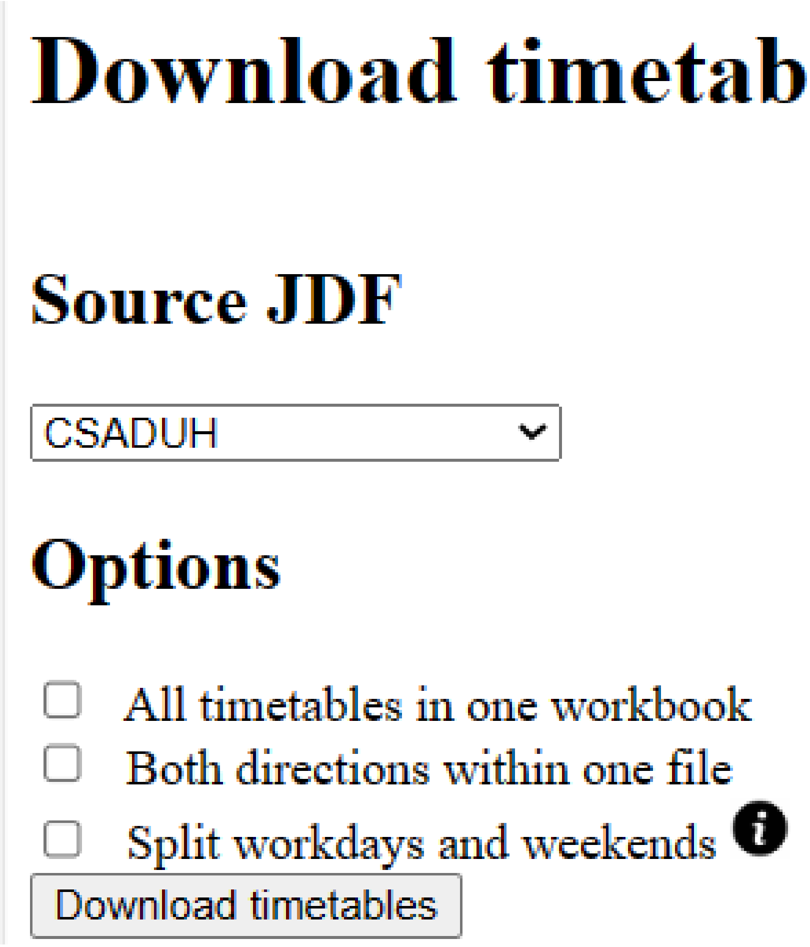

Download timetables
Source JDF
CSADUH
Options
□ All timetables in one workbook
□ Both directions within one file
□ Split workdays and weekends
Download timetables

Figure 8.9 Timetable exporting page

dropdown menu
and click the *Download timetables* button.

The interface is shown in the figure 8.9.

8.7.1 Export settings

By default, each timetable (one line and one direction) is created into an individual worksheet (of XLSX file) and then sent as a zip archive with name identical to the JDF batch. The name of the file and worksheet is `<Line number>_<Line version>_<Direction>.xlsx`. The direction is either F or B.

The settings allow for these changes, when checked on:

- *All timetables in one workbook* - all timetables are merged into one workbook.

However, it is also sent as zipped for consistency.

- *Both directions within one file* - the timetables for both directions are merged into one worksheet. The name of the worksheet will not then contain the direction.

- *Split workdays and weekends* - this does not change the file structure, but the trips in the timetables will be listed first for working days, then for weekends.

After the timetables are made, the zipped file is immediately sent to the user (download).

8.7.2 Schema of the exported file

The output format is XLSX, which can be opened in multiple spreadsheet applications, such as LibreOfficeCalc[2] or Apache OpenOffice[3].

The actual file schema is described visually on the image 8.10.

8.8 Upload of timetable in XLSX format

This module, available under the title **Upload XLSX**, allows to upload timetables on the server in XLSX format to be automatically parsed and converted to JDF, in the same way as the JDF batch would be uploaded directly (see 8.3).

It is meant to only support the files in similar format as exported from the TTP application, which also means only some data is preserved.

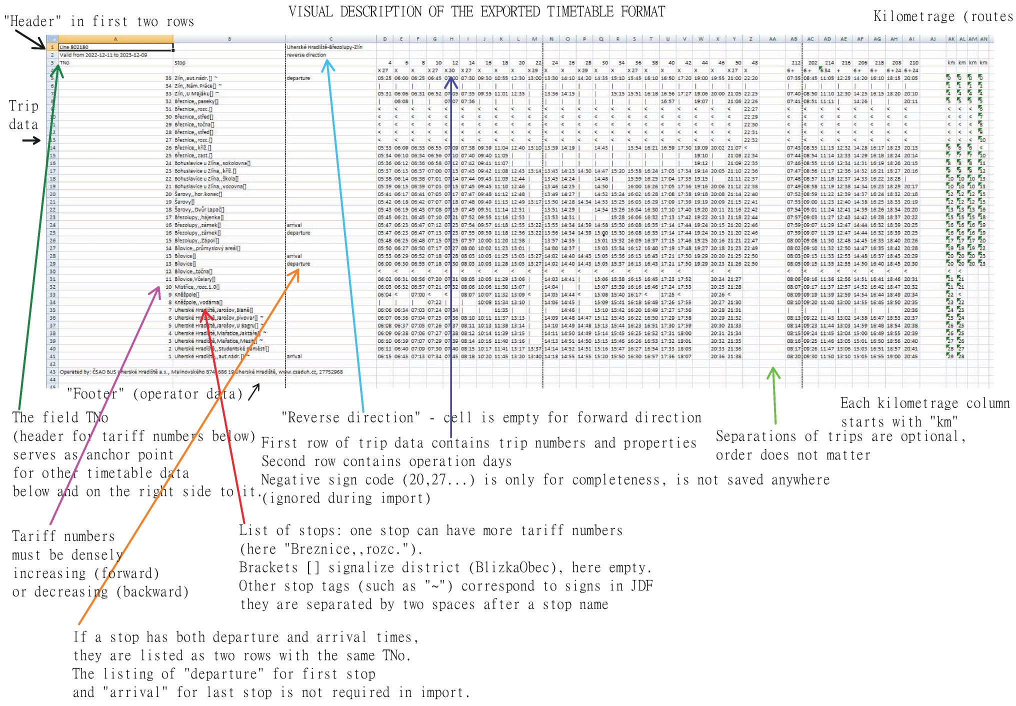
The input form accepts only one XLSX file without any additional settings. In order to import multiple lines, simply include them in the same file. This corresponds to the option *All timetables in one workbook* in the export settings (see 8.7).

Uploading the exported timetables will preserve the trip data, but strip away time codes (trips (not) operating on specific dates). However, the periodic days of operation (`1-7,X,+`) in the trip metadata are preserved, which makes the trip querying and scheduling calculation behave rationally for common days. [4]

[2] https://www.libreoffice.org/discover/calc/

[3] https://www.openoffice.org/

[4] Tip: to calculate scheduling for specific days, simply delete the trips not operating on these days before upload. On the other hand, to guarantee operation on any date, remove the periodic sign.



VISUAL DESCRIPTION OF THE EXPORTED TIMETABLE FORMAT

"Header" in first two rows

Kilometrage (routes)

Trip data

	A	B	C
1	Line 802180		Uherské Hradiště-Březolupy-Zlín
2	Valid from 2022-12-11 to 2023-12-09		reverse direction
3	TNo	Stop	
4			
5	35	Zlín,,aut.nádr.[] ~	departure
6	34	Zlín,,Nám.Práce[] ~	
7	33	Zlín,,U Majáku[] ~	
8	32	Březnice,,paseky[]	
9	31	Březnice,,rozc.[]	
10	30	Březnice,,střed[]	
11	29	Březnice,,točna[]	
12	28	Březnice,,střed[]	
13	27	Březnice,,rozc.[]	
14	26	Březnice,,křiž.[]	
15	25	Březnice,,zast.[]	
16	24	Bohuslavice u Zlína,,sokolovna[]	
17	23	Bohuslavice u Zlína,,křiž.[]	
18	22	Bohuslavice u Zlína,,škola[]	
19	21	Bohuslavice u Zlína,,vozovna[]	
20	20	Šarovy,,hor.konec[]	
21	19	Šarovy[]	
22	18	Šarovy,,Dvůr Lapač[]	
23	17	Březolupy,,hájenka[]	
24	16	Březolupy,,zámek[]	arrival
25	16	Březolupy,,zámek[]	departure
26	15	Březolupy,,Zápolí[]	
27	14	Bílovice,,průmyslový areál[]	
28	13	Bílovice[]	arrival
29	13	Bílovice[]	departure
30	12	Bílovice,,točna[]	
31	11	Bílovice,Včelary[]	
32	10	Mistřice,,rozc.1.0[]	
33	9	Kněžpole[]	
34	8	Kněžpole,,vodárna[]	
35	7	Uherské Hradiště,Jarošov,Blaně[]	
36	6	Uherské Hradiště,Jarošov,pivovar[] ~	
37	5	Uherské Hradiště,Jarošov,U Bagru[] ~	
38	4	Uherské Hradiště,Mařatice,Jaktáře[] ~	
39	3	Uherské Hradiště,Mařatice,Mesit[] ~	
40	2	Uherské Hradiště,,Studentské náměstí[]	
41	1	Uherské Hradiště,,aut.nádr.[] ~	arrival
42			
43	Operated by: ČSAD BUS Uherské Hradiště a.s., Malinovského 874, 686 19 Uherské Hradiště, www.csaduh.cz, 27752968		

"Footer" (operator data)

The field TNo (header for tariff numbers below) serves as anchor point for other timetable data below and on the right side to it.

"Reverse direction" - cell is empty for forward direction

First row of trip data contains trip numbers and properties
Second row contains operation days
Negative sign code (20,27...) is only for completeness, is not saved anywhere
(ignored during import)

Separations of trips are optional, order does not matter

Each kilometrage column starts with "km"

Tariff numbers must be densely increasing (forward) or decreasing (backward)

List of stops: one stop can have more tariff numbers (here "Breznice,,rozc.").
Brackets [] signalize district (BlizkaObec), here empty.
Other stop tags (such as "~") correspond to signs in JDF they are separated by two spaces after a stop name

If a stop has both departure and arrival times, they are listed as two rows with the same TNo.
The listing of "departure" for first stop and "arrival" for last stop is not required in import.

Figure 8.10 Visual example of the exported XLSX file

8.9 Visualizing trips and schedules on map

The visualization of trips and schedules is available in the navigation under the title **Visualize trips and schedules on map**.

In order to use this, two prerequisites need to be satisfied:

1. A schedule needs to be prepared (see 8.5)

2. The stops used in the schedule must be uploaded and their locations set (see 8.6)

The form takes three input fields. The first input field is for the schedule upload to be selected from the file system (presumably created by the **Schedule buses** module).

The second input field is a dropdown for how to visualize connection between two stops: either via a direct line, or by more accurately drawn route (using the OSM navigation data).

The third input field is a dropdown how to visualize multiple trips with overlapping routes.

8.9.1 Output files

When submitted, the server will create two maps per each bus block contained in the schedule. They are all saved as zip file and sent to the user. One of the maps (with extension `.html`) is meant to be viewed in the browser showing the route on OSM background (requires internet connection), the other (with extension `.pdf`) is a PDF image of a created plot, where the coordinates correspond to latitude and longitude.

8.9.2 Data interpretation

For each bus block, only data which refer to stops visited by the bus are relevant. If a trip goes through multiple stops (common occasion), it will be considered as several trips in this section.

In this section, the word *line* is also meant as a literal line on the map, not with the timetable context.

The stops are connected by a line, which is colored based on the time of the trip. The colors follow rainbow, with trip starting at 00:00 being red and trip starting at 23:59 being violet.

If there is only one trip which connects two stops and an option *Direct line between two stops* is chosen, such trip will be displayed as a straight line between two stops. However, if an option *Use OSM navigation* is chosen, the line will attempt to follow the actual road network (as if more nodes were added between the stops).

If some stops in the schedule are missing on the map, they are left out and a line is simply drawn between the previous and next stop.

The line thickness can vary, based on the speed of the trip (simply calculated as time between two stops divided by the distance), thicker line means higher speed.

8.9.3 Overlapping lines

If multiple trips share the same route (in either direction), the resolution depends on how the option *Draw trips going through the same stops as* is set. This also applies for the road network nodes if the option *Use OSM navigation* is chosen.

For the most concise option, select *Overlapping lines*. The overlapping routes will be drawn as a multi-colored line segment (along its length), keeping its original width. The more colors are used, the more trips share the same route.

To show the trip density better, select the option *Parallel lines*. Each trip will be drawn as offset from the previous one, the central line will be directly connecting the stops (assuming odd number of trips).

An alternative is to select *Curves*, which renders the trip between two stops as arcs with varying radius, but ensuring the arcs always start and end at the stop locations.

In general, multiple trips between two stops in one direction are rendered ascending by their time, with trips in the opposite direction following, as can be seen on the figure 8.11.

Trips going from west to east (increasing longitude) are considered as the forward direction, and trips going from east to west as the backward direction.

8.10 File structure of the saved data

The temporary data saved by the server application is stored in the `online_files` directory. This is the root directory for the data.

The directory `upload` contains the uploaded JDF batches. The stops locations are saved in the file `stops_locations.csv` in the root.

The directory `distances` contains the distance matrix files, which are further divided into subdirectories based on the method used for calculation (map,timetable,upload).

Other files, which can be directly re-generated from the requests, are saved in the `temp` directory. These are again divided into subdirectories. Note the data do not get automatically deleted, only overwritten.

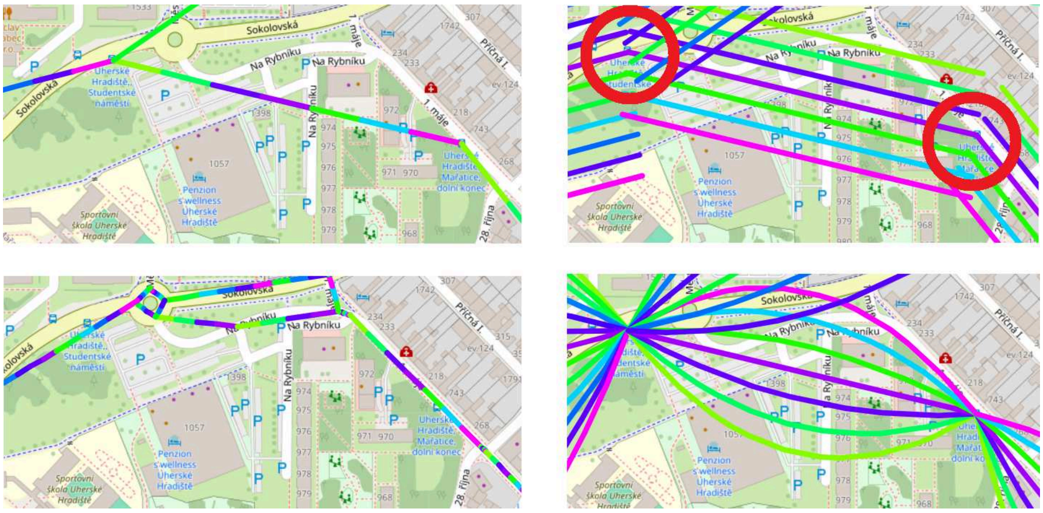

Sokolovská
Na Rybníku
1. máje
Příčná I.
28. října
Uherské Hradiště, Studentské náměstí
Penzion s wellness Uherské Hradiště
Sportovní škola Uherské Hradiště
Uherské Hradiště, Mařatice, dolní konec

Figure 8.11 The difference in visualization. Left top - direct lines with overlapping trips as changing colors. Right top - parallel lines (relevant stops highlighted by red circle, not shown on the actual output). Right bottom - curved lines. Left bottom - using OSM navigation data (overlapping trips as changing colors). From the color change between 3rd and 4th line, we can see there are 3 trips in west-east direction and 4 in the opposite.

All of these files have `.json` extension.

- `departures` - departures from one stop (name based on stop name)
- `arrivals` - arrivals from one stop (name based on stop name)
- `trips` - trips in given JDF, generated during the scheduling preparation. These data are read again when the schedule is actually generated. [5] The name is based on the JDF batch name and the date of operation.
- `schedules` - the calculated schedules. The name is based on the JDF batch name, date range, distance matrix method, and scheduling method (goal).

[5] This allows for rather hacky modification by replacing the file with custom trips, right before the schedule is submitted.

Chapter 9

Installation and launching

For local server, follow these instructions as for standard Python program:

9.1 Prerequisites

The TTP was tested on Linux and Windows. A Python 3.10 interpreter is required.

9.2 Source codes and installation

1. Extract the source codes from the attached ZIP file
   TTP.zip and extract them.

2. Go to the folder TimetableBusScheduling (where the extracted files are located).

3. Set up Python environment and install the required packages by running:

   ```
   python3.10 -m venv venv && . venv/bin/activate
   python3.10 -m pip install -r requirements.txt
   ```

4. Launch the server by interpreting file Flask_Main.py from the folder TimetableBusScheduling. As for the port, select a number such as (5000) that is not already in use.

   ```
   python3.10 Browser_Interface/Flask_Main.py -p <port>
   ```

5. You can also use flag -d to run the server in developer mode.

6. Launch your browser and go to 127.0.0.1:<port> to use the app.

7. If needed, the server can be stopped by pressing Ctrl+C in the terminal.

9.3 Data retrieval

The datasets used in this thesis are available in the attached ZIP file `Datasets.zip`. After extracting, they can be uploaded to the server by using the **Upload JDF** module in the TTP.

The locations of most stops relevant to those datasets are already included in the server filesystem. To add more, follow the user documentation (Section 8.6).

9.4 Pre-loaded data

The data created by the server will be stored in the `online_files` folder. This folder is also provided separately in another ZIP file `online_files.zip` where the data generated during the testing run is already included. You can compare the results of your run with the provided data.

9.5 Remote server

To use the server on remote PC, use port forwarding, e.g.

```
ssh -L 5000:localhost:5000 user@remote_server
```

and install everything on the remote server as described above. You can then access the server from your local browser.

Chapter 10

Programmer documentation

The whole TTP is written in Python and is separated into four packages. One module is responsible for browser interface (frontend) to the TTP, the other three are internal logic packages: JDF conversion, Bus scheduling and Map visualization.

The user interface uses Flask framework for working on browser. Each submodule is a Flask blueprint responsible for handling a small amount of pages (1-2). In order to be correctly displayed in browser, all data are displayed to the user as HTML (with Jinja templating).

It is assumed that the users are patient enough for the application to load the data, therefore asynchronous communication is not used.

Originally, the TTP was developed as a console application, where the modules worked independently, outputting text files, which were then processed by the next module. The interconnection, which is done via the browser in the final version, therefore also mixes business logic with the presentation layer.

10.1 Timetable processing

The module `Timetable_Calculations`
in the package `JDF_Conversion` is responsible for loading all JDF data into individual in-memory objects and performing calculations which can be inferred from them. The data are loaded directly from filesystem, without any database engine.

For the TTP application, the main purposes of this module are:

1. Loading the JDF data into memory.

2. Calculating the shortest path between two stops.

3. Determining operation days of trips.

4. Merging multiple JDF data into one.
5. Loading and exporting Excel files representing timetables.
6. Extracting trips to be passed to bus scheduling.

10.1.1 JDF classes

The whole timetable processing module is designed to load the whole JDF batch into memory and use it as a source for further calculations (as opposed to a relational database). This approach has an advantage of easily visible data structure when debugging, a major disadvantage is a memory consumption and also speed.

Each class starting with `Jdf` in the file `JDF_Classes.py` represents data in JDF batch. For example, an instance of `JdfDopravce` objects corresponds to one row in the file `Dopravci.txt`.

In case of the classes and its fields, it can be justified by the fact the JDF 1.11 (whose data are used) uses the Czech names for the fields, so the names in TTP are chosen in the same way to correspond.

Apart from the JDF 1.11 standard fields, there are also some additional fields to simplify calculations. The constructors of each class simply takes the values from the CSV file and assigns them to the fields. A method `Serialize` then does the opposite - it takes the object and returns a list (corresponding to one CSV row).

More can be seen in section 10.1.4. The method `Bind` is responsible to create in-memory references to other objects. For example, `JdfSpoj` (trip) has a reference to `JdfLinka` (line) - this is a N:1 relationship. These references are made by the corresponding JDF fields (each trip has a line number in the CSV). However, the application also allows for 1:N references, here it would be multiple trips assigned to one line. For these, the variable ending on `Coll` (as collection) is used in the `Bind` method.

Not all JDF objects are fully completed in the code. This is mostly due to focus scope of the TTP, also some data are not widely used in the publicly available JDF batches (e.g. "Oznacnik" field in `JdfCasSpojLinkaZastavka`).

Some values can contain only enums. In case of 0/1 values,they are converted to booleans during initialization. If there are more values (like in `JdfCasKod`), the ranges of values are stored in the module `Timetable_Enums`.

10.1.2 JDF data initialization

The class representing a JDF batch is in the module `Timetable_Calculations`. Its method `LoadJDF` (with folder name passed as parameter) loads individual JDF data using the module `csv`. The internal method `_loadSingleJdf` is modified to handle JDF 1.10 files as well (the parameter objType is used to determine the class to be used, and if it fits the JDF 1.10 standard, additional empty fields are added to the row). Ultimately, this batch is held in an instance of the class `JdfProcessor`, where also additional fields are included for easier processing.

The module is designed to allow for loading multiple JDF batches at once, which is necessary as the initial data contain one line per batch, which would not allow for good optimization and data aggregation.

For this, a class `JdfMerger` was created, which merges individual `JdfProcessors`, and the final data binding (creating in-memory references across trips and stops) is done in the method `FinishMerge`. The merging process is described in the section 10.1.3.

The method `ParseSingleFolder` serves as an entry point for initializing a JDF processor from a single batch. It takes a name of folder containing the JDF batch (text files as described in the JDF documentation), and returns an instance of JDF processor, or `None` in case of failure.

For multiple folders, an analogous method `ParseMultipleFolders` is used. It is not supposed to crash if an invalid batch is passed, it gets simply skipped (error is printed on console, see method `JdfMerger.AddNew`).

10.1.3 JDF merging

The goal of JDF merging is to have one batch for more lines. Since the original data contain one line version per batch, the TTP assumes the lines in input data will be different.

If given batches are completely disjoint - the trips take place in different cities, the merging is straightforward.

The main issues for correct merging are stops and lines. While stops might have different identifiers, they can be considered the same if they have the same name. This requires to change the field ‘CisloZastavky‘ on a given stop so both processors use the same number. That transitively causes the change of identifier in other objects which refer to this stop number. The second merging issue arises when multiple versions of one line appear in different batches. This is simply resolved by incrementing the line discriminator field (`RozliseniLinky`), but this requires update of all references to this line, which are mostly contained in trips (`JdfSpoj`) and stop events(`JdfCasSpojLinkaZastavka`), see the method `ChangeKey` which is written to handle both stops and lines (although in completely different way).

After all potential JDF processors are merged, the function `FinishMerge` returns an instance of `JdfProcessor` with all the data merged and expanded (all references created).

Using serialization as can be seen in 10.1.4 then can save the JDF into a file, so the next time they can be loaded all at once.

10.1.4 JDF serialization

The module `JDF_Serialization` contains function for JDF data serialization. The most relevant function for TTP is `SerializeJdfCollection`, which is parametrized by the collection (of stop objects, trip objects, etc) and output file. For collections sent from JDF processor created from single JDF batch, the output is identical as in the original files. The serialization is done by calling the `Serialize` method on each object in the collection (as the JDF objects should have it defined).

The functions `PackDeparture`, `PackArrival` and `PackTrip` are used for dictionary representation of trips (etc.) as an alternative for string representation (these representations are then passed to the frontend, see 10.5.5).

10.1.5 Timetable algorithms

With purely JDF data loaded, two algorithms are implemented in the module `Timetable_Calculations`:

- Approximate distance between two stops
- Listing of trips operating on a given day

To calculate an approximate distance between all stops, a method `GetAllStopMatrixByTT` (of the class `JdfProcessor`) is used. It first builds a graph in-memory (edge lengths represented by `numpy` array) with the `CalculateTimeMatrix` method, which checks all trips and determines the stop distances on given trips (direct connections). Since one route is shared by multiple trips, the shortest time difference is always picked (regardless of day time). After such matrix is built, Dijkstra's algorithm implemented in the `scipy` library is used to find the actual closest distances between stops in the graph. Due to time zone issues, as a hotfix, the graph is be disconnected regarding trips across state borders (this is in timetable represented by the `CLO` sign). It is assumed the TTP will be used for regional transport, therefore the timezone issue is basically ignored (and might need fixing). In regard to the bus scheduling, a method `GetDeadheadMatrixByTT` is used for returning only distance matrix between terminal stops. The function therefore returns a list of stops (list of names) and a distance matrix (as `numpy` array).

For listing trips operating on a given day, a method `CheckTripsInDay` is used. Each trip is checked in the function `IsTripOperated`. The function directly writes the trips into a provided text stream.

10.1.6 Excel timetable export

The module Table_Export is responsible for exporting the timetable data to XLSX format. First, the timetable is loaded into a 2D list of strings using three functions:

1. MakeTimetable - creates a 2D list of trips from the JDF data (trips, stops, stop times).

2. CompleteTimetableMetadata - adds line number, line operator, to the table

3. AddTimetableKilometrage - adds kilometre columns to the table

The function WriteAsExcel then writes this table to an Excel file using the xlsxwriter library. The function ExcelTimetables is responsible for calling all these functions in a succession. It takes JdfProcessor as input and creates the Excel file, where each line in the processor is represented as a pair of two sheets in a workbook. The line name is in format: <Line number>_<Line version>_<Direction>. Direction is T for forward and Z for backward. orientation (depending on the tariff numbers).

On the other hand, a function ZipTimetables creates one workbook per each line (and version) which contains two sheets - one for each direction. All workbooks are then zipped into one file using zipfile library.

10.2 Bus scheduling

Here we actually calculate the bus schedules from provided data, which have already gone through the timetable processing. The module therefore supplies all needed functions for the optimizing problem itself and also allows visual rendering on console.

All source files for this module are saved in the folder Bus_Scheduling.

10.2.1 Data representation

The classes are saved in a file Scheduling_Classes.py. An object representing a trip is described in the class Trip. It consists of line and trip number, then starting and ending stop (as strings), and starting and ending time (as strings). The field "Via" optionally allows for more representation, as dictionary of stops on the way (including starting and ending) with respective stop times. Only stops actually served in the trip are considered.

For easier processing, the starting and ending time are first loaded from JSON as strings, while converting the strings to minutes (relative to midnight) is done in the method RecalcMinutes, which also allows to add relative starting day (so the starting and ending time will be for example offset by 1440 if we plan to start scheduling from 1.1.2023 but the trip starts at 2.1.2023).

As to represent pull-outs, pull-ins and deadheads, for this is used class `Deadhead`, which also uses starting and ending stop. Unlike trip times for departure and arrival, here we use Duration (in minutes) and earliest time of arrival on the stop. It is used only to simplify the data representation when writing out the schedule.

The scheduling also uses a distance matrix to show deadhead times. In file system, it is represented as tab-separated file with headers (they are processed by the `csv` library), while in the code, it is represented as `numpy` 2D array regarding the distances (always integer) and stops are as simple list; index of a stop corresponds to the row in array.

In order for quicker stop index retrieval, also a stops map is created, which simply maps the stop name (string) to its index.

10.2.2 Precalculations

In the bus scheduling, we assume the deadhead times are already calculated externally. The method `CreateFeasibilityGraph` allows us to retrieve the feasibility (compatibility) relation between trips, represented by tuples (indices of respective trips). It checks if there suffices enough time to get from the last stop on LHS trip to the first stop of RHS trip (see 5). A custom attribute `maxWaitHours` can be used to prune edges which would cause too long waiting time between two trips. This might not yield an optimal solution in the sense of bus amount, but this is not researched (todo: let's try).

The method `EstimateTripsNoDepot` could be used for estimating the time needed for calculations, but this would require more research, so currently it returns only trips and edges count as a simple dictionary.

The edges are also assigned weights in the function `CreateDeadheadMap`. For parametrization, we can use `waitingPenalty` (scaling with waiting time between trips, deadhead time not included in the waiting).

10.2.3 Scheduling algorithm - entry point

The bus scheduling from other modules is called from the function `CalculateGeneral`, which takes all the input data (trips,stops,distance matrix) plus arguments for how should the scheduling be optimized (parameter `schedulingMethod` as string enum and optionally scheduleArgs). The schedules are then simply returned as jagged array of trips (represented by the class Trip).

10.2.4 Scheduling algorithm - default

The algorithms are saved in file Bus_Scheduling.py. For a simple optimization, a library networkx is used, as the default solutions can be obtained by a help of graph algorithms.

A function OptimizeTripsOptionalDepot takes a list of trips, matrix of distances, and dictionary of stops to their indices.

The algorithm is briefly described in 5: two vertices are created per trip (departure vertices have the same number as respective trip index, arrival vertices have their number as trip index increased by length of the trips). The vertices are not added explicitly, if a trip is not on either side of the feasibility relation (=among the edges), it will not be added to the graph.

After a matching is calculated, the function convertEdgesToBuses creates feasible bus schedules based on the matching. Unmatched trips are considered as well by adding an extra bus which covers only them. This includes trips which were not in the compatibility graph at all (e.g. very late trip with high restriction on waiting time).

If it is set to false, the optimization continues with respecting the deadheads. Two vertices are added per each bus and these are connected with other vertices (first vertex per bus is connected to all departure vertices, second vertex is connected to all arrival vertices) by edges representing zero length pull-out and pull-in. This allows us to always find minimum weight full matching (as the full matching will always exist).

10.2.5 Scheduling algorithm - single depot

A graph scheduling algorithm with single depot uses the same function as default algorithm, but this time the edges representing pull-out and pull in have also specified weight. To get an exact depot name, the function get_close_matches from difflib library is used - we allow to only specify depot as an existing stop. This allows to use the same distance table as we use for deadheads.

10.2.6 Scheduling algorithm - linear programming

The scheduling algorithm with single depot also has its linear programming version OptimizeTripsSingleDepotLP, which uses mip package with CBC solver.

10.2.7 Scheduling algorithm - circular, approximate

The approximate scheduling uses the default scheduling as base algorithm (accepting the same parameters) optimized by minimizing deadheads. Additional parameters (all integers) refer to local search parameters:

1. `iterations` - how many times do we eliminate unfit solutions
2. `kept` - how many solutions will remain after each elimination (max)
3. `multiplications` - amount of times each solution is improved upon

The function `EvaluateLength` evaluates the total deadhead time of a given solution (including returning times) as a "fitness" function.

The local search (see 5.6) uses a `Swap` operation, which is described by 4 parameters:

1. `Bus1I` - index of first bus block
2. `trip1I` - index of last trip of first bus before splitting
3. index of second bus block
4. index of last trip of second bus before splitting

An index `-1` for the trip means the first part of the bus block will be empty (the whole block will be moved to the other bus block).

The swap operation is then applied as in this Python code:

```
buses[bus1I] = buses[bus1I][:trip1I+1] + buses[bus2I][trip2I+1:]
buses[bus2I] = buses[bus2I][:trip2I+1] + buses[bus1I][trip1I+1:]
```

Each of the feasible solutions is then described using these fields:

1. `swap` - operation (see above) to be done on the solution (or None)
2. `buses` - list of bus blocks
3. `length` - fitness of the solution (minimize)
4. `swappable` - list of all feasible swaps

New solutions are created in two phases: We first choose `multiplications` random swaps per each solution, then we evaluate the fitness using `EvaluateSwapDiff` without actually swapping the trips, creating a new copy with updated `length` and `swap` field.

After we generate all new solutions, we eliminate the unfit ones and then use the method `ApplySwap` for remaining solutions, where a deep copy of the `buses` list and swappable is created and the swap is applied (updating the list of bus blocks and the new valid swaps).

This allows us to save memory by applying the swap only after the solution is actually picked for the next iteration. The set of all valid swaps does not need to be recalculated fully, only the swaps which apply to one of the previously swapped bus blocks.

An index -1 refers to an empty first part of the bus block, so all original trips are moved to the other bus block.

Creating a new solution picks a random bus and for each bus it tries to find all pairs of edges which can be swapped: For example, if a bus A has consecutive trips $(1, 3, 5)$ in its block, and bus B has consecutive trips $(2, 4, 6)$ in its block, then the algorithm might check swapping on edges $(3, 5)$ and $(2, 4)$, leading to block $(1, 3, 4, 6)$ for bus A and a block $(2, 5)$ for bus B - for this, it needs to be considered if edges $(3, 4)$ and $(2, 5)$ are feasible.

The rescheduling always swaps the full right hand side of the respective bus blocks (right side of the swap list area), which alters only the deadhead at the swapping place and the final turnaround time.

For this, the function EvaluateSwapDiff is used, which shows how much would the total time changed without having to recalculate both blocks.

For eliminating unfit solutions, first only random solutions are selected, then the best remaining solutions are selected based on their fitness. The currently optimal solution is always kept.

A function SelectRandomWeightedIdx, which is unused in the current implementation, could be used for selecting random solutions as a roulette wheel selection.

10.2.8 Schedule formatting on console

For printing on console, the schedules are rendered using prettytable and the sources are in Schedule_Rendering.py.

The function FormatSchedule takes the schedule and adds pull-outs,pull-ins and deadheads. The scheduled trips are passed as a jagged array (list), where each subarray represents a block of a bus. It also requires using distance matrix and list of stops, which are used for adding deadheads. Deadhead is not inserted if its length is zero (consecutive ending and starting stop are the same). The parameter addFinalDeadhead causes the function to add a deadhead from the last stop to the first stop of the block. If a depot is included, its pull-out and pull-in times are included as well.

It returns two lists: first are bus blocks with included deadheads, second are total deadhead times per bus.

The function TableSchedules converts each schedule to actual table with respective headers (times as integers are now converted to hh:mm string format). The function PrintTabularSchedules then takes these tables and prints them to console using pretty-printing.

10.2.9 Schedule format as JSON

The methods mentioned below are always working with Python dictionaries, which are then converted to JSON using json library.

The schedule contains all bus blocks with also some metadata about how the scheduling was created (arguments passed).

The methods of creating schedules are defined in Schedule_Rendering.py as well. The function SchedulesToJsonDict converts the schedule to JSON format. Arguments related to bus scheduling are at the top level (most importantly the method used, e.g. "method":"default"). The function currently also ignores the field "cached", this is a taken dependency from the frontend, where the schedule can be cached from already existing file if user requests it multiple times.

The bus blocks are saved under the key Bus blocks,containing bus number and then trips.

The formatting of a trip object to dictionary is done
in the method ToDictForJson for a Trip object. It contains of these fields (if not specified, format is mandatory string):

- Day - day of operation (format “YYYY-MM-DD“)
- Line - line number (integer)
- Trip - trip number (integer)
- Start stop - starting stop
- End stop - ending stop
- Start time - departure time from first stop
- End time - arrival time to last stop
- Stops - dictionary (string:string) of stops on the way with respective times (optional)

The reverse parsing (from file to schedule) is done
in the function SchedulesFromJsonDict, which returns the blocks and schedule arguments.

10.3 Map visualization

This module connects the TTP to OpenStreetMap related modules which allows to see the rendition of trips on the map. In this section, the word "line" is used in a geometric sense (more precisely, as a line segment). Be sure to pay attention to the coordinate naming. While alphabetically they are ordered as x, y, the coordinate y corresponds to latitude and x to longitude.

10.3.1 Line plotting - utility functions

For correct line plotting, some simple utility functions like `GetLineLength` or `PerpendicularBisector` are used, utilizing common formulas from analytic geometry. If the parameter is a tuple, then the first element is always the x coordinate and the second element is the y coordinate.

10.3.2 Route plotting overview

In a file `OSM_Distances.py` a function `PlotSchedules` is responsible for whole generation and saving of whole schedules (all bus blocks at once). The parameter `trackComplexity` shows if the stops for respective trips should be connected with a direct (or curved) line, or if the plot should directly follow the road (more granularity). The "following the road" is based on OpenStreetMap data (accessed by `osmnx`) library, where the granularity is obviously not perfect - generally the accuracy of data depends on the accuracy of OSM . The whole usage of OSM library is discussed in other section.

For visualization, all stops per each trip should be displayed on the map. A function `ExpandTrips` takes one bus block (as list of dictionary, described in the section of Bus scheduling) and all the stops on the way as to create list of separate trips, so instead of e.g. 2 trips with 10 stops each, a bus now will have 20 trips with 2 stops.

When the trips are expanded, each bus block is processed sequentially. Each trip within the block has a pair of stops, creating a sequence of x and y coordinates. These coordinates are taken from the dictionary stopLocations, which maps stop names to coordinates as tuple.

If a departure stop from a trip does not match the arrival stop of the last trip, a deadhead is created. This gets represented by changing the line style to dashed. Note that the lines are not created yet, the function only creates four lists - `xs,ys,times,styles,tripNos`.

If the track is supposed to be represented realistically (not only direct connection of stops), a function `InterpolateRoutes` is called , which will add extra points between the stops, while keeping the same style.

Once all the coordinate and time data are ready, the route is plotted. A figure using module `matplotlib` is created, then filled using the function `PlotRoutes`, which is described in the next section. and after rendering, the map is saved using module `mplleaflet`, leading to a HTML file to be saved which allows to see the plot (routes) on OSM background.

10.3.3 Route plotting algorithm

The main function for plotting routes is `PlotRoutes`. It uses `pyplot` module and plots the route on axes. The function takes the following parameters:

1. `figure,axes` - axes to plot on (from `pyplot`)
2. `xs,ys` - coordinates of stops on route (as lists of floats)
3. `times` - times of arrival to stops (as list of minute integers)
4. `styles` - styles of lines between stops (as list of two values)
5. `sameRoutes` - how to plot identical routes so they can be visualized when overlapping (string enum)
6. `thicknessMin,thicknessMax` - thickness of line based on speed between two stops

It tries to check which routes are identical (very small difference between x and y coordinates) and plot them in a way that they can be distinguished. The exact explanation for used algorithms is written as comments in the code, no external functions for determining the additional location of points is used. After the points are adjusted, the method `plot_axes` is called until all routes get processed. (It does not return anything, only modifies the plot.)

A module `cm` is used for color distinction of trips based on times they commence.

10.3.4 Leaflet map rendering

The package `mplleaflet` is used for converting the `pyplot` plots for rendering on a OSM background. As a newer version of `matplotlib` is used, the `mplleaflet` package was not directly compatible with it. This required a small change of the code in the `mplleaflet` package itself, as can be seen on a [GitHub issue]. Therefore, the TTP carries this modified version of the package in the folder `mpllf_remake`.

10.3.5 Using OSM navigation

The file `OSM_Distances.py` heavily uses modules `osmnx` and `networkx` to retrieve data from OpenStreetMap and then apply functions allowing us to calculate correct driving times or to correctly render navigation data for visualizing routes. First of all, out module (`OSM_Distances`contains many utility functions used to retrieve correct coordinates from bounding box. The function `ChooseDownloadArea` takes a list of points and then returns the 4 bounding coordinates based on the points (northernmost,southernmost, etc.). These values then can be passed to function `ExpandBoundingBox`, which can expand this area by given amount of kilometers, and finally to `DownloadOSMDataBox`, which returns the OSM data for the given area.

The OSM data are returned as a graph, allowing to directly apply `networkx` library functions. In order to find routes between stops, sometimes not the exact point corresponding to a stop could be found - we must find a point on a road. For this, the graph is projected and corresponding point on a road is found by naive algorithm.

10.3.6 Finding stops on map

For correct visualization, it is also needed to find the stops on the map.This feature is not integrated into the app, but used as independent script. as they are supplied by the config file (which was created with help of this module). Although these data are directly supplied by configuration file, it was used for quick initial searching.

This script is used for finding the coordinates of stops on the map. It uses the Overpass API to query the OpenStreetMap database. The query is mostly fixed with user providing the queried areas and their administrative values. All stops within the specified areas are then queried and mapped on user-provided stops with approximate names. If these names do not fit, a city centre is chosen instead.

In the file `Stops.py`, the class `StopWithLocation`
is used to represent a stop as a union of its name and location. The fields `Obec,CastObce,BlizsiMisto,BlizkaObec` (to correspond with Czech names) are used to represent the stop name, stringified as format "1,2,3[4]" (trailing commas removed). The method `parseName` is used to create the 4 parts from the stringified name, whereas the method `getName` is used to create the stringified name from the 4 parts. There can be multiple formats based on the parameter `blizkaObecMode` (see comments).

A function `SanitizeStopName` is used to standardize the stop name, by removing spaces and treating extra commas as separators. This functionality

should not be needed if the stops are correctly named
or fixed from the external script fix_stop_names.py.

To save stop locations from OSM persistently,
a function GenerateStopLocationFile is used. The parameter jsonFiles is a list of files of the json dumps, and the output (into outPath) is written as CSV data. The function can have 4 modes, each mode defines what happens in the old stops for the output file. In all cases, the new and (potentially) old data are sorted.

10.4 Stop searching by OSM query

A file StopsSearcher.py contains functions for searching stops directly on a map.

The function FindStopsInArea calls a Overpass query to find all bus stops in given area name with administrative level derived from supplied area type. The argument areaType is based on Czech area classification to derive area type. The queried data are represented as nodes (from OSM) and then converted to dictionary with these keys:

- id - identifier of the stop (integer)
- lat - latitude (float)
- lon - longitude (float)
- tags - tags of the stop (dictionary, including name - see documentation for OSM)

If they are to be output to file, they are naturally saved as JSON.

The function FindRespectiveStopsInfo is its wrapper for assigning stop data to stops by their names. Both stop names and area names are list of strings. Running an overpass query is done by copying area names (no sanitization for SQL injection is done, responsibility of the user).

While it might happen that sometimes stop names are not accurate, the library difflib is used for finding the closest stop names as they are in the OSM. As a fallback mechanism, the location of the municipality centre is used. Totally, 2 queries are therefore ran.

The output is then a dictionary mapping a stop name to its relevant OSM information as described in 10.4.

10.5 User interface

The user interface is written in HTML and uses Flask framework for routing and rendering. It provides access to the other submodule functions and allows to see the results of the calculations.

A default routing use case is as such:

1. A selected page is requested - GET request
2. User submits a form from given page - POST request, redirecting to an ‘api/‘ endpoint
3. The ‘api/‘ endpoint calls the appropriate function and redirects to the GET request of a new page.

Some alternative data are kept in session variable, if they were unwieldy to pass as GET parameters.

10.5.1 General structure

The entry point of the interface (and basically the whole TTP) is at the `Flask_Main.py` file (see 10.5.3). This file includes all other submodules using Flask blueprint system.

Each other file (starting with `Flask_`) then registers its blueprint with the suffix `_Api`. The blueprint is then used to define the routing of the submodule. The methods within a submodule are first ordered as routing with GET requests, then POST requests, then other helper functions alphabetically.

For displaying HTML content, Jinja templates are used. They are stored in the folder `templates` and are rendered by the Flask engine (calling function `render_template`). That method takes the name of the template file and optional variables to be passed to the template, and returns string. Returning string from the decorated function then automatically returns a HTML response to the user’s page.

The pages (as result of rendering the templates) are in standard HTML5 (with CSS - mostly inline - and Javascript where appropriate).

The extensions `datatables` and `fontello` are saved locally in the `static` folder.

Error handling within the application is mostly delegated to helper functions, which return True/False as their first argument, and then the response/error as the second argument.

In case of failure, the TTP usually redirects to the relevant module (directly reachable from main menu) with error messages included in a template (see file `inline_errors.html`).

Functions which require some arguments usually do not have separate function for argument validation, everything is mostly done at a beginning of respective function.

The `root` variable serves within the application as global variable for the root directory of the application.

When referring to a specific location, we will use the format `/module/page` as in the code, the full location would be e.g. `127.0.0.1:5050/module/page`

10.5.2 Template format

All templates are saved in the `templates` folder. Other included files are saved in the `static` folder.

Some templates are included in multiple other templates; the typical beginning of templates served to the user looks like this:

```
{%include 'header.html'%} //HTML header with js/css imports
<body>
{% set title='Title' %} //Page title to highlight
{% include 'navigation.html' %} //Navigation bar
{% include 'inline_errors.html' %} //Error message
```

10.5.3 Main Flask file

Here gets initialized configuration of the application and session. A method `app.register_blueprint` is used for including other submodules related to the interface. The only functions defined in the main file are for the index page and for calculating total request time (which is printed only on console for measurements).

10.5.4 JDF processing

The original JDF processor assumed the JDF data to be kept in local memory. Since this is harder to do for browser based applications, the JDF processor object is recreated every time a request for processing JDF batch data is sent.

10.5.5 Trip, departure, arrival querying

In the `Query_Api` submodule, the default page (at '`/query/main`') shows the form the user can use for quering trips, departures, or arrivals during speicfic date. The form is then submitted to the '`/query/result`' endpoint as a GET request. A JDF folder specified by the argument is unpacked, then other arguments are validated and then passed to the functions `queryTrips`,`queryArrivals` and queryDepartures. Since for more effectivity the user can query 3 categories (trips, departures, arrivals) at once, in case of a failure an error only for relevant category is added, while other categories continue being processed.

As the JDF processing API returns the trips (or departures, arrivals - we will not repeat this later) in non-table format, the function unpackTrips converts the data, so they can be passed to the template query_result.html as simple list of lists.

For example, in case of list querying, the initial format could be expressed in type hint syntax as List[Tuple[Day,List[Trip]]] (Day is a string, Trip is a dictionary), and the converted format would then add the Day field to each of the trip, so the resulting format is then List[List[Trip]].

Currently, the fields of the Trip object (dictionary) are as following:

1. Day - (queried) day of operation (format "YYYY-MM-DD")
2. LineNo - line number (integer)
3. TripNo - trip number (integer)
4. StopFrom - starting stop
5. StopTo - ending stop
6. TimeFrom - departure time from first stop (format "hh:mm")
7. TimeTo - arrival time to last stop (format "hh:mm")

When extending these, they should also be extended in the related template.

A local constant maxAmount is used to limit the amount of returned results within each category (the limit is checked after all trips for particular day are queried, so more than this limit can actually be returned).

10.5.6 JDF upload and merge

The blueprint Upload_Api is used for serving request related to adding more JDF batches to the server by user. TThe route /upload/main simply shows the form uploading_form.html for uploading the JDF folders. The option for uploading more folders at once is handled by JavaScript.

The form is then submitted as POST request to route /upload/main/submit. The function submitFolder then takes all these folders as list of files and tries to reassign them the tree structure (we can take an assumption when the user uploads multiple folders, they will have different name, so the files will have names such as F1/Zastavky.txt,F2/Zastavky.txt, etc.), if a user uploads multiple folders under same name, the earlier files will be overwritten.

Each folder is then processed individually under the assumption it contains a JDF batch and then also merged, as described in the documentation for JDF processing.

The whole result on which folders (batches) got succesfully uploaded, which were corrupted, overwritten, etc. is saved as session variable (under the key `request_dict`):

1. `doCopy` - if the folders should be uploaded, not only merged (bool string)
2. `goodNew` - list of folders which were succesfully uploaded and were not yet on the server
3. `badNew` - list of corrupted folders
4. `goodOverwrite` - list of uploaded folders which replaced the existing ones
5. `badOverwrite` - list of corrupted folders, old folders under this name still exist on server
6. `doMerge` - if the folders should be merged (bool string)
7. `mergeName` - name of the JDF batch to be created by merging
8. `mergeOverwrite` - if the merge did overwrite existing JDF batch (bool string)
9. `mergeOk` - if the merge was successful (bool string)

There is no checking for the event if the user wants to do a merge of only one batch - this should be prevented on client side, and is well-defined on server side.

After the upload, the user is redirected to the site `/upload_result`, where the respective information about uploaded folders is extracted from the session variable and displayed in the template `upload_result.html`. A class `UploadReport` is used to bundle the information about the upload (only simply with information text, names of batches, and the color based on the success).

The colors are standard CSS colors, so they can be used directly in the template `upload_result.html`.

10.5.7 Management of stops

The blueprint `Stops_Api` is used for adding location of the stops on the server. The file location of stops themselves is described in the variable `stopsFile`.

The file is a CSV representation of the stop objects described in the section 10.3.6.

The display of stops is available on the route /stops, which uses the template stops_list.html. This template includes a Leaflet library (as JavaScript) for showing a map background.

The rendered page also contains two forms (POST method) for editing stops:

Importing new stops directly from OSM is available on the route /stops/import, which uses the function GenerateStopLocationFile as described in the subsection 10.3.6.

Clicking on the map to create new stops, or checking the mark for deleting stops is tied to a form which is handled on a route /stops/edit. Regarding the deletion of stops, a client sends list of numbers to the server. They represent the index of stop in the displayed list.

Both handling routines then update the stop list file and redirect again to /stops.

10.5.8 Bus scheduling

In the blueprint Scheduling_Api, the route /schedules/form is used for displaying the form for scheduling. The rendered form is in the template schedule_form.html, sending a POST request on /schedules/prepare.

A submission of the form itself does not create schedules yet, only causes the TTP to calculate trips relevant for given scheduling range and a distance matrix which are then cached as files - the exact names are based on the JDF name and date, as seen in the documentation. Based on the distance matrix and requested trips, an estimate from Bus_Scheduling module is made. It has been reduced to simply showing amount of vertices (trips) and edges (feasibility relations) in the graph. The location of these files, as well as other scheduling parameters are stored in session variable.

This causes the redirection to location /schedules/preview where the data are again taken from the session variable, so upon user's submission, they are again sent as GET parameters on the route /schedules/submit ,since the data are now easily put into a query string.

The whole validation is done in the function handleScheduling, and after the scheduling is done, the user is redirected to the route /schedules/result, where the results are displayed in the template schedules/result.html. Similar to the trip querying, the scheduling API returns schedules as list of lists, so the function unpackSchedules is used to flatten them.

The schedules are also cached (as JSON) in a file, whose name is created in the function createScheduleFileName.

The datatables used for schedules allow expansion. It should be noted that the datatables library does not allow a convenient way to load data into it without using AJAX, so as a work-around, the trip data are saved in the JavaScript variable tripsData, and then loaded into a table using format_trips function when expanding the row.

This causes lots of ballast in the HTML file, but all together, it is not larger than one of the JSON files with trips for one day, as the maximum schedule date range is 24 hours.

10.5.9 Timetable export

The blueprint Timetable_Api allows the user to download timetables from uploaded JDF batches. The route /timetables/export/main displays the form for selecting JDF batch and download options as described in Timetable calculations module. The XLSX timetables are always created from JDF data, they are never cached, and therefore immediately sent to the client.

10.5.10 Map visualization

The blueprint Visualization_Api is used for visualizing bus schedules on the map, as described in chapter 6. The route /visualize displays the form for selecting schedule file, using the template visualize/form.html. The form is submitted to the route /visualize/submit as POST request (as file upload is needed), which then redirects the user to route /visualize/result. The plots are temporarily saved in directory whose name is based on the schedule file, but at the end of processing, they are all zipped and the temporary folder is removed.

For possible future multiprocessing, the folder created for the map visualization has appended random id at its name, as the processing of files might take too long time, as to prevent overwriting (unlike e.g. schedules, where there is only one file created per request).

The plots are displayed in two variants - one is HTML (with Leaflet background), the other is PDF as default Python plot with labels.

10.5.11 File downloading

The blueprint Download_Api is used for downloading files from the server. Here get redirected all requests for downloading files from appropriate modules (the path always starts with /download):

1. `/download/schedules` - for downloading optimized bus schedules (moduleScheduling_Api).

2. `/download/distance_matrix` - for downloading distance matrix between terminals in given JDF batch (module `Scheduling_Api`).

3. `/download/visualization` - for downloading schedules visualized on map (module `Visualization_Api`).

The functions use method `resolve` for getting the actual location of a file (absolute path), as there were some issues with paths relative to the root.

Chapter 11

Conclusion

11.1 Result

We have managed to write a software that should be able to help with the workflow of public transport analysis:

- Loading JDF data or importing them from XLSX.
- Exporting the data to human-readable timetable format.
- Showing the departures and arrivals from given stops.
- An approximate algorithm for finding bus assignment to respective trips.
- Displaying the data about bus scheduling.
- Visualising the routes on a map.

We have shown that the software works correctly on reasonably large datasets with replicable results.

11.2 Difficulties

We can also list the main difficulties that we have encountered:

- The capabilities of converting the XLSX data to the JDF data are generally limited, due to the lack of standardization and therefore the need to implement our own validity checks.
- Another issue with JDF format is that stop locations are not supported, so they need to be supplied externally.

- The algorithm for handling OSM data requires a lot of memory and time, making the TTP struggle with larger regional transports.
- The web interface implementation might make this software hostable online, but it would need to have many safeguards against denial of service caused by the slow “business logic“.

11.3 Future work

The future work could be focused on the following areas:

- More comfortable direct data exchange between the user and the server (e.g. scheduling for arbitrary trips via user interface)
- More advanced bus scheduling algorithm
- Faster and richer map visualisation
- More advanced timetable import/export without the loss of comfortability
- Driving time warnings for unrealistic timetables (based on navigation data)
- Using GTFS format for timetable data

11.4 Final words

The TTP aims to cover multiple areas in the field of public transport analysis. It probably cannot measure up to the commercial software in terms of performance and comfortability, but is able to handle the default task and offers many features available for further development.

Bibliography

[1] Czech Republic. *Zákon č. 194/2010 Sb. o veřejných službách v dopravě.* URL: https://www.zakonyprolidi.cz/cs/2010-194.

[2] Barbora Berečková. *Alternativní přístupy k výběrovým řízením na zajištění dopravní obslužnosti.* Jan. 2019. URL: https://dspace.cvut.cz/handle/10467/80659.

[3] Avishai Ceder and Nigel H.M. Wilson. "Bus network design". In: *Transportation Research Part B: Methodological* 20.4 (1986), pp. 331–344. ISSN: 0191-2615. DOI: https://doi.org/10.1016/0191-2615(86)90047-0. URL: https://www.sciencedirect.com/science/article/pii/0191261586900470.

[4] Philine Schiewe. *Integrated Optimization in Public Transport Planning.* Jan. 2020. ISBN: 978-3-030-46269-7. DOI: 10.1007/978-3-030-46270-3.

[5] A A Bertossi, P Carraresi, and G Gallo. "On some matching problems arising in vehicle scheduling models". In: *Networks* 17.3 (1987-01). ISSN: 0028-3045. DOI: https://doi.org/10.1002/net.3230170303.

[6] Vitali Gintner, Natalia Kliewer, and Leena Suhl. "Solving large multiple-depot multiple-vehicle-type bus scheduling problems in practice". In: *OR Spectrum* 27 (2005), pp. 507–523. DOI: https://doi.org/10.1007/s00291-005-0207-9.

[7] Czech Republic. *Zákon č. 111/1994 Sb. Zákon o silniční dopravě.* URL: https://www.zakonyprolidi.cz/cs/1994-111.

[8] MobilityData. *General Transit Feed Specification.* 2024. URL: https://gtfs.org/ (visited on 05/04/2024).

[9] Python Software Foundation. *Python.* Version 3.10.13. URL: https://www.python.org/ (visited on 05/04/2024).

[10] Armin Ronacher. *Flask.* Version 2.3.2. URL: https://flask.palletsprojects com/ (visited on 05/04/2024).

[11] Czech Ministry of Transport. *Metodický pokyn k organizaci CIS JŘ*. 2014. URL: https://www.mdcr.cz/getattachment/Dokumenty/Verejna-doprava/Jizdni-rady,-kalendare-pro-jizdni-rady,-metodi-(1)/Jizdni-rady-verejne-dopravy/metodicky-pokyn-cis-5.pdf.aspx.

[12] CHAPS spol. s.r.o. *Portál jízdních řádů*. 2023. URL: ftp://ftp.cisjr.cz/JDF.

[13] Jan Masopust. *Automatické zpracování českých jízdních řádů autobusů*. Tech. rep. 2020. URL: https://www.researchgate.net/publication/348752371_Automaticke_zpracovani_ceskych_jizdnich_radu_autobusu.

[14] Microsoft Corporation. *MS-XLSX: Excel (.xlsx) Extensions to the Office Open XML SpreadsheetML File Format*. 2024. URL: https://learn.microsoft.com/en-us/openspecs/office_standards/ms-xlsx/2c5dee00-eff2-4b22-92b6-0738acd4475e (visited on 05/04/2024).

[15] Radek Papež. *Aplikace strojového čtení souborů JDF*. 2023. URL: https://portal.radekpapez.cz.

[16] Seznam.cz a.s. *Mapy.cz*. 2023. URL: https://mapy.cz/.

[17] Avishai Ceder. *Public transit planning and operation: Modeling, practice and behavior*. CRC press, 2016. ISBN: 978-1-138-31307-6.

[18] Stefan Bunte and Natalia Kliewer. "An overview on vehicle scheduling models". In: *Public Transport* 1.4 (Nov. 2009), pp. 299–317. DOI: 10.1007/s12469-010-0018-5. URL: https://doi.org/10.1007/s12469-010-0018-5.

[19] John E. Hopcroft and Richard M. Karp. "An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs". In: *SIAM Journal on Computing* 2.4 (1973), pp. 225–231. DOI: 10.1137/0202019. URL: https://doi.org/10.1137/0202019.

[20] Marek Peřina. *Problematika bezpečnostních přestávek řidičů při optimalizaci oběhů vozidel*. June 2019. URL: https://dspace.cvut.cz/handle/10467/83242.

[21] Czech Republic. *Nařízení vlády č. 589/2006 Sb.* URL: https://www.zakonyprolidi.cz/cs/2006-589.

Appendix A

External scripts

This is just a brief documentation of using helper scripts to handle JDF data and stop locations. The scripts are written in Python and are attached to the package with TTP source code.

A.1 JDF merging

The script `merge_jdf.py` is used to merge multiple JDF batches into one.

The functionality is equivalent to the merging option in 8.3.

The script takes the following arguments:

1. variable number of unnamed arguments — paths to the folders with JDF batches to merge.
2. last argument — path to the folder where the merged JDF batch will be saved.

An example of usage:

```
$ venv/Scripts/python.exe merge_jdf.py \
    ../Datasets/Havirov_Split/havirov_*
    ../Datasets/Havirov_MHD
```

A.2 Stop name standardization

The script `fix_stop_names.py` allows to standardize stop names, especially for a case of urban transport.

As in the case of the JDF merging script, (A.1), the initial arguments are lists of JDF folders, while the script expects only the Zastavky.txt file in each of them (this option was chosen for more comfortable processing).

The script directly overwrites the Zastavky.txt files in the input folders.

The default behavior only resolves commas inside the quoted text fields (if the field for city name contains a comma, the rest of the field after the comma is resolved as a city part).

Other arguments (added after the folder names) are:

- `-c` or `--city` — adds a city name to the stop name, if it is not already there. Useful for urban transport which leaves the city name out of the stop name.
- `-e` or `--exclude` — used with the previous option, to not add city name to stops starting with this city name. Useful when the urban transport includes other cities than its central one.
- `-d` or `--district` — adds a name of nearby city (=district - fourth part of the stop name) to the stop name, if it is empty.
- `-p` or `--place` — for stops with 2-field names, moves the second field (city part) to the third field (location) Make sure to fix false hits after using this option - for stops which are supposed to contain city name and city part, but not place.

An example of usage:

```
$ venv/Scripts/python.exe fix_stop_names.py \
    ../Datasets/Havirov_MHD \
    -c "řHavíov" -d "KA" \
    -e "Šenov" "Horní Suchá"
```

A.3 Stop location finding and map visualization

The file `Map_Visualization/OSM_Distances.py` can be ran directly to find the location of stops using Overpass API, and to visualize the schedule on a map.

In case of import errors, a path might be needed to exported properly, such as (on Windows bash shell):

```
$ PYTHONPATH=. venv/Scripts/python.exe \
   Map_Visualization/OSM_Distances.py <args>
```

As a pre-requisite, a bus schedule file in JSON format is needed. It can be obtained from running the scheduling module in TTP (see section 8.5) using the timetable data in order to calculate the schedule.

The first two arguments are positional:

1. path to the JSON file with the bus schedule

2. path to the file with stop location data, can already exist

Other mandatory arguments:

- `-r` or `--regions` — select named areas (regions) where to find the stops.
- `-a` or `--admin` — administrative level[1] of the searched region. In case of the Czech Republic, a suitable area type is “Kraj“, which has an administrative level of 6 (this is also a default).

Optional arguments:

- `-l` or `--load` — load the stop location data from the file instead of querying the API, if such stop data already exists.
- `-s` or `--stops-only` — do not visualize the schedule, only find the stop locations.
- `-m` or `--map-dir` — directory to save the map files (if `-s` is not used).

The output file with location data can be directly imported in the TTP (see section 8.6).

An example of usage:

```
$ PYTHONPATH=. venv/Scripts/python.exe \
    Map_Visualization/OSM_Distances.py \
    "C:\Downloads\Havirov_timetable_default.json" \
    "StopsInfoArriva.json" \
    -s -d "Moravskoslezský kraj" -a 6
```

[1] https://wiki.openstreetmap.org/wiki/Key:admin_level