ASSIGNMENT OF BACHELOR’S THESIS

Title: Mobile Application for Route Search in Prague Public Transport
Student: Artem Kandaurov
Supervisor: RNDr. Jiřina Scholtzová, Ph.D.
Study Programme: Informatics
Study Branch: Computer Science
Department: Department of Theoretical Computer Science
Validity: Until the end of summer semester 2018/19

Instructions

DPP Offline is a mobile application for the iOS platform, which finds public transport routes in Prague using open GTFS data. However, the application does not always find an optimal solution. Many received feedbacks refer to that the application often finds routes with unnecessary line changes or does not offer the best lines to the destination. It is obvious that the problem is the ineffective algorithm of searching routes. The aim of the thesis is to design and implement a new version of the application with respect to limited power of mobile devices. The new version, like the old one, should be able to find routes without an Internet connection.

1) Analyze existing solutions, discuss advantages and disadvantages.
2) Revise the used algorithm and optimize the data structures in the recent solution.
3) Design and implement a new version of the application, give emphasis to used algorithm and data structures.
4) Propose and discuss some user enhancements and how they could be implemented.

References

Will be provided by the supervisor.
Bachelor’s thesis

Mobile Application for Route Search in Prague Public Transport

Artem Kandaurov

Department of Theoretical Computer Science
Supervisor: RNDr. Jiřina Scholtzová, Ph.D

May 14, 2018
Acknowledgements

Foremost, I would like to thank my supervisor RNDr. Jiřina Scholtzová, Ph.D. for all the valuable advice and a lot of time spent on this project. I am also grateful to my friend Ing. Eduard Alibekov for the original idea of creating such application and interesting discussions on the topic of shortest path algorithms in time-dependent graphs. And, of course, I want to thank my wife, Ekaterina, for her care and unlimited moral support.
Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 14, 2018
Czech Technical University in Prague  
Faculty of Information Technology  
© 2018 Artem Kandaurov. All rights reserved.  
This thesis is school work as defined by Copyright Act of the Czech Republic.  
It has been submitted at Czech Technical University in Prague, Faculty of  
Information Technology. The thesis is protected by the Copyright Act and its  
usage without author’s permission is prohibited (with exceptions defined by the  
Copyright Act).

Citation of this thesis

Kandaurov, Artem. Mobile Application for Route Search in Prague Public  
Transport. Bachelor’s thesis. Czech Technical University in Prague, Faculty  
Abstrakt

Tato práce se zabývá problémem nejkratších cest v časově závislém váženém multigrafu. Příkladem takového grafu je síť veřejné dopravy postavená s GTFS daty. Cílem práce je navrhnout strukturu dat a algoritmus pro vyhledávání tras v dopravních sítích a optimalizaci pro zrychlení procesu vyhledávání tras. Výsledkem je offline mobilní aplikace pro vyhledávání tras hromadné dopravy v Praze.

Klíčová slova  Veřejná doprava, Časově závislý graf, Algoritmy nejkratších cest, GTFS
Abstract

This thesis deals with the shortest path problem in a time-dependent directed weighted multigraph. An example of such graph is a public transport network built with GTFS data. The aim of the thesis is to design data structure and algorithm for routes search in transport networks and design optimizations to improve route search process. The result is an offline mobile application for public transport route search in Prague.

Keywords Public transport, Time-dependent graph, Shortest path problem algorithms, GTFS
## Contents

**Introduction** 1

1 **Theoretical Introduction** 3  
1.1 Terminology 3  
1.2 Algorithm 4  
1.3 GTFS Data 4  
1.4 Data Structure 5

2 **Analysis of Existing Solutions** 7  
2.1 Existing Software Products 7  
2.2 Conclusion 9

3 **Analysis of the First Version** 11  
3.1 Data Transformation and Preprocessing 11  
3.2 Routes Search Algorithm 13  
3.3 Conclusion 14

4 **Design** 17  
4.1 Routes Search Algorithm 17  
4.2 Data Structure 21  
4.3 Conclusion 23

5 **Implementation** 25  
5.1 Application Structure 25  
5.2 Route Search Algorithm 25  
5.3 Data Structure 28  
5.4 Functions 29  
5.5 Distribution 30  
5.6 Conclusion 30
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>User Interface of IDOS</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>User Interface of PID Info</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>User Interface of CG Transit</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>User Interface of DPP Offline</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Data structure of the database used in the first version of DPP Offline</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>Dijkstra (left) versus Bidirectional Dijkstra (right) dispersion</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Revised data structure. Relationships mean foreign keys. One arrow means &quot;to-one&quot; and double arrow means &quot;to-many&quot; relationships.</td>
<td>22</td>
</tr>
<tr>
<td>6.1</td>
<td>Route search comparing. The first version (left) versus the new version (right). Route Kublov – Vozovna Žižkov.</td>
<td>34</td>
</tr>
<tr>
<td>6.2</td>
<td>Route search comparing. The first version (left) versus the new version (right). Route Dívčí hrady – Kajetánka.</td>
<td>34</td>
</tr>
<tr>
<td>6.3</td>
<td>Route search comparing. The first version (left) versus the new version (right). Route Terminál 3 – Hotel Golf.</td>
<td>35</td>
</tr>
<tr>
<td>6.4</td>
<td>Route search comparing. The first version (left) versus the new version (right). Route Zličín – Bílá Hora.</td>
<td>35</td>
</tr>
</tbody>
</table>
List of Tables

1.1 Required GTFS files ........................................... 4
1.2 Optional GTFS files ........................................... 5

2.1 Competitiveness analysis of all four applications. Average search
time has significant errors in measurement and are presented for
basic understanding of the competitiveness situation. Database
size for IDOS and PID Info could not be defined because the
databases are stored on the servers .................................. 9

3.1 Description of data structure of the database used in the first ver-
sion of DPP Offline .................................................. 12

4.1 Description of revised data structure .......................... 22

5.1 Comparing algorithms under the same conditions. In the table are
given time intervals spent to find a route .......................... 28

6.1 Comparing the first version and the new version of DPP Offline
under the same conditions. In the table is given time spent to find
a route .............................................................. 33
Introduction

Many city dwellers use public transportation and want to have all needed information about the journey in their pockets, which is why transport mobile applications are so popular. However, there is an enormous amount of data representing public transport networks of modern cities, it is not easy to process this data on mobile devices. For example, the Prague public transport network contains more than one thousand stops and more than one and a half million departure times.

Most mobile applications for searching public transport routes do not solve the problem of processing big data on mobile devices. These applications require Internet connection and calculate necessary data using server computation power. Often, such applications are simply covers for the essential solutions on the server side. However, there is another type of application that only uses device resources. These applications do not require a permanent Internet connection and use advanced algorithms and data structures to find routes in fractions of a second.

One of those mobile applications is DPP Offline. It was released in 2016 for iOS by the author of this thesis. The application is intended for Prague’s public transport network. Unfortunately, the implementation of this application has a lot of problems and left much to be desired from the user point of view. Feedback often referred to how the application often found routes with unnecessary line changes or did not offer the best lines to the destination. Moreover, the time needed to calculate a route is much longer than in other similar applications.

It is obvious that the main problems are ineffective route search algorithm and inappropriate data structure. The main goal is to design and implement a new version of DPP Offline for route search in Prague that takes into consideration the limited power of mobile devices. The new version, like the old one, should be able to find routes without an Internet connection. Furthermore, the new version must find more convenient routes and work much faster than its predecessor.
Chapter 1

Theoretical Introduction

The application has two sensitive areas which can affect final accuracy and processing time. These areas are data structure and algorithm for routes search. The algorithm has a direct effect on route search performance and result route optimality. The data structure provide access to the needed data and indirectly affect the running speed of the algorithm. This chapter introduces problems and terminology considered in the thesis.

1.1 Terminology

First, terms from graph theory must be briefly defined. A lot of information is obtained from $\mathbb{1}$. Graph is an ordered pair $G = (V, E)$, where $V$ is a set of vertices and $E$ is a set of two element subsets of $V$. Set $E$ names the edge set and represents relationships between pairs of vertices. A multigraph is a graph that can have more than one edge between a pair of vertices. A directed graph is a graph where every edge from set of edges $E$ is an ordered pair of vertices. A weighted graph is a graph where all edges have an assigned numerical value by weight function. A time-dependent graph is a graph that utilizes a cost function $\text{cost}(e, t)$ where $t$ is the current time. This function returns the cost of an edge depending on the time at which the query is executed. A graph which describes a public transport network is time-dependent directed weighted multigraph.

Some public transport terms within this thesis must also be defined. All stops with the same names form a node. A line (e.g. tram number 15) is a set of trips with the same name that passes the same sequence of nodes. Most often a line has two directions, but there may be more or less. Departure is defined by line, departure time, source, and target stops. Departure is unique and represents an edge between two nodes in transportation network. Trip is a consistently ordered sequence of departures that belongs to the same line. Simply explained, trip represents all departures in a line’s iteration. Finally,
1. Theoretical Introduction

A route is an ordered sequence of departures that forms a path between two nodes. Each next departure in the route must start from the previous one’s target. All these terms are described also in article [3].

1.2 Algorithm

The algorithmic problem considered in this thesis applies to a subset of the shortest-path problems. The basic shortest-path problem is defined as the problem of finding a path between two vertices in a graph such that the sum of the weights of the path’s constituent edges is minimized [4].

The problem can be specified as finding an ordered sequence of departures which form a route in a time-dependent, directed, and weighted multigraph such that the route is optimal to the given criteria.

1.3 GTFS Data

Data of public transport schedules is needed to construct a transportation network. Prague Public Transit Company releases this data for free in GTFS format [5]. All transportation data used in the application is built from this data. The expiration time of the provided data is one week, then the data becomes invalid and must be updated. However, updating is rarely a problem because the new data will be provided in the same GTFS format.

The GTFS format was developed by Google in 2005 and is a collection of CSV files with all data about the transportation network and associated geographic information. These files have defined structures that allow for use without needing to contact the transport agency. Table 1.1 reviews files required in the GTFS format [5].

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agency.txt</td>
<td>One or more transit agencies that provide the data in this feed.</td>
</tr>
<tr>
<td>stops.txt</td>
<td>Individual locations where vehicles pick up or drop off passengers.</td>
</tr>
<tr>
<td>routes.txt</td>
<td>Transit routes. A route is a group of trips that are displayed to riders as a single service.</td>
</tr>
<tr>
<td>trips.txt</td>
<td>Trips for each route. A trip is a sequence of two or more stops that occurs at specific time.</td>
</tr>
<tr>
<td>stop_times.txt</td>
<td>Times that a vehicle arrives at and departs from individual stops for each trip.</td>
</tr>
<tr>
<td>calendar.txt</td>
<td>Dates for service IDs using a weekly schedule. Specify when service starts and ends, available week days.</td>
</tr>
</tbody>
</table>

Table 1.1: Required GTFS files.
1.4. Data Structure

The GTFS format describes optional files that can be provided by transport agencies. These files are described in Table 1.2. In the case of Prague Public Transit Company, the agency provides two optional files: shapes.txt and calendar_dates.txt.

Table 1.2: Optional GTFS files.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shapes.txt</td>
<td>Rules for drawing lines on a map to represent a transit organization’s routes.</td>
</tr>
<tr>
<td>calendar_dates.txt</td>
<td>Exceptions for the service IDs defined in the calendar.txt file.</td>
</tr>
<tr>
<td>fare_attributes.txt</td>
<td>Fare information for a transit organization’s routes.</td>
</tr>
<tr>
<td>fare_rules.txt</td>
<td>Rules for applying fare information for a transit organization’s routes.</td>
</tr>
<tr>
<td>frequencies.txt</td>
<td>Headway (time between trips) for routes with variable frequency of service.</td>
</tr>
<tr>
<td>transfers.txt</td>
<td>Rules for making connections at transfer points between routes.</td>
</tr>
<tr>
<td>feed_info.txt</td>
<td>Additional information about the feed itself, including publisher, version, and expiration information.</td>
</tr>
</tbody>
</table>

More information about the GTFS format can be found e.g. in the FIT CTU master’s thesis.

1.4 Data Structure

Data structure has a strong influence on resulting performance. Every request to the data structure must be processed as soon as possible, so the main criteria in the choice of data structure are the result size and data fetching speed.

The problem can be defined as a finding of a suitable data structure format (e.g., SQLite, raw) and its content to reach the optimal ratio between result size and data fetching speed. This data structure must contain all data from the original GTFS files needed for route search and represent transportation network in the form of a graph.
Analysis of Existing Solutions

Transport applications are very popular in the current era of intelligent mobile devices. In this chapter, four main competitors were selected to compare their parameters. All four competitors provide systems for public transport route search in Prague and are available on the App Store for the iOS platform.

2.1 Existing Software Products

All statistical data in this section is collected by the Apptrace service. The information presented here is correct as of 30th April 2018.

- IDOS

The most popular application for public transport routes search in the Czech Republic is IDOS. The application has a lot of useful features and quickly finds wanted connections. It allows searching for only direct routes (without line changing) and setting minimal time for transfer. Subjectively, this is the best application if the user has permanent access to the Internet; however, it has advertisements. The iOS version of the application was launched in 2011.

Figure 2.1: User Interface of IDOS.
2. Analysis of Existing Solutions

- PID Info [1]

PID Info is the official application from PPTC and was launched for the iOS platform in 2016. PID Info, as IDOS, requires permanent Internet connection. This application has not gained much popularity even though PPTC advertises it inside transport and at stations. The possible reason for this could be that the first version of the application (named DPP INFO) had many bugs. The most recent version is much better and has become competitive. The application allows for limiting the number of transfers and the choice of transport types for searching.

Figure 2.2: User Interface of PID Info.

- CG Transit [2]

This is the only application (except DPP Offline) that allows users to search for routes in Prague without a permanent Internet connection. The data provider for this application is Chaps [3]. This application contains all the features offered by rival applications even though it works without an Internet connection. The first release for the iOS platform was in 2011. Unfortunately, the application is pay-to-use and users must buy an annual license to use it without restrictions.

Figure 2.3: User Interface of CG Transit.
• DPP Offline

The thesis focuses on DPP Offline. This application was released in 2016 for the iOS platform. It is distributed free of charge and does not require a permanent Internet connection. The first version received negative feedback about the application’s functioning, which is what triggered this thesis. This application is described in detail in chapter 3.

Figure 2.4: User Interface of DPP Offline.

Presently, DPP Offline’s only competitor is CG Transit, which has the same philosophy but is pay-to-use. Table 2.1 lists the competitiveness analysis of all four applications. All tests were performed with the Apple iPhone SE.

Table 2.1: Competitiveness analysis of all four applications. Average search time has significant errors in measurement and are presented for basic understanding of the competitiveness situation. Database size for IDOS and PID Info could not be defined because the databases are stored on the servers.

<table>
<thead>
<tr>
<th></th>
<th>IDOS</th>
<th>PID Info</th>
<th>CG Transit</th>
<th>DPP Offline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free of charge</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Offline mode</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Other cities</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Internal map</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Contains advertising</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Average search time</td>
<td>800 ms</td>
<td>1 sec</td>
<td>600 ms</td>
<td>6 sec</td>
</tr>
<tr>
<td>Database size</td>
<td>–</td>
<td>–</td>
<td>11 MB</td>
<td>205 MB</td>
</tr>
</tbody>
</table>

2.2 Conclusion

In this chapter four application were compared. There are another applications, that help to search public transport routes, e.g. Google Maps [14], but route search is not their main function. So, route searching in such applications is implemented poorly.
As depicted in table 2.1, DPP Offline has many weaknesses. Internet independence is the most important feature of DPP Offline, but CG Transit has implemented this feature more successfully. Moreover, Internet independence takes a lot of work because the route search algorithm and data structure must be fast enough to present results in a fraction of a second using the limited computing power available to a mobile device.
Chapter 3

Analysis of the First Version

Other than DPP Offline, none of the other applications has open source code to process implementation analysis. Hence this chapter only reviews the first version of DPP Offline. The whole application is written in Objective-C [15]. This version was developed in the summer of 2016 with open GTFS data from the PPTC. The algorithms and data structures used in the version are described below.

3.1 Data Transformation and Preprocessing

For faster access to GTFS data, it has to be translated to the suitable data structure. The relational SQLite database [16] was chosen to be the format for the data structure, because the application was developed for the iOS platform and the SQLite format is compatible with Apple’s Core Data [17]. A database structure was designed to fit the GTFS data and is mapped in figure 3.1 and described in table 3.1.

Figure 3.1: Data structure of the database used in the first version of DPP Offline. Relationships mean foreign keys. One arrow means ”to-one” and double arrow means ”to-many” relationships.
3. Analysis of the First Version

Table 3.1: Description of data structure of the database used in the first version of DPP Offline.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stop</td>
<td>identifier</td>
<td>Unique identifier of the stop.</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>Name of the stop.</td>
</tr>
<tr>
<td></td>
<td>lat</td>
<td>The latitude of the stop.</td>
</tr>
<tr>
<td></td>
<td>lon</td>
<td>The longitude of the stop.</td>
</tr>
<tr>
<td></td>
<td>node</td>
<td>Node associated with this stop.</td>
</tr>
<tr>
<td>Node</td>
<td>identifier</td>
<td>Unique identifier of the node.</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>Name of the node.</td>
</tr>
<tr>
<td></td>
<td>formattedName</td>
<td>Node.name attribute without diacritics.</td>
</tr>
<tr>
<td></td>
<td>arrivals</td>
<td>Array of departures that arrives to the node.</td>
</tr>
<tr>
<td></td>
<td>departures</td>
<td>Array of departures that departure from the node.</td>
</tr>
<tr>
<td></td>
<td>stops</td>
<td>Array of associated stops.</td>
</tr>
<tr>
<td>Departure</td>
<td>identifier</td>
<td>Unique identifier of the departure.</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>Name of the line to which the departure belongs.</td>
</tr>
<tr>
<td></td>
<td>days</td>
<td>Integer mask with information about days of week when the departure is working.</td>
</tr>
<tr>
<td></td>
<td>startDate</td>
<td>Date from which the departure is working.</td>
</tr>
<tr>
<td></td>
<td>endDate</td>
<td>Date to which the departure is working.</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>Departure time.</td>
</tr>
<tr>
<td></td>
<td>minutesInWay</td>
<td>Minutes from departure to arrival.</td>
</tr>
<tr>
<td></td>
<td>nodeFrom</td>
<td>The node from which the departure leaves.</td>
</tr>
<tr>
<td></td>
<td>nodeTo</td>
<td>The node to which the departure arrives.</td>
</tr>
<tr>
<td></td>
<td>previousDeparture</td>
<td>The next departure on the same trip.</td>
</tr>
<tr>
<td></td>
<td>nextDeparture</td>
<td>The next departure on the same trip.</td>
</tr>
<tr>
<td>Matrix</td>
<td>data</td>
<td>Binary data of the Floyd–Warshall matrix</td>
</tr>
<tr>
<td></td>
<td>isNight</td>
<td>Flag indicating if the matrix is prepared for day or night.</td>
</tr>
</tbody>
</table>

Not all of the GTFS data provided by PPTC was used in the first version of the application. Data from files `agency.txt` and `shapes.txt` were not processed, which is why data from this file was not part of the database structure. However, the structure contained all the needed data to find routes...
The database has linking properties that help to speed up access time. For example, to find the next or previous departure on the same trip, the algorithm should not search in the whole database because it is enough to use the `Departure.previousDeparture` or `Departure.nextDeparture` attributes. Moreover, the database contains two preprocessed properties to reduce computation in the runtime. The first attribute is `Node.formattedName`, that stores `Node.name` without diacritics for faster stop searching. A lot of people search for their needed stop by writing its name without diacritics. The second attribute is `Departure.minutesInWay`, which replaces arrival time from the original GTFS data.

This database is adapted for routes search with Floyd–Warshall path matrices (more at section 4.1.1). The entire database is filled simply by parsing GTFS files, except for the `Matrix` entity. Calculating $O(|V|^3)$ in the runtime would be complicated, so path matrices are pre-calculated and stored in the database. These matrices create with algorithm 1 when the rest of the database is filled.

In this implementation, the database contains two path matrices: day and night versions. This decision was made because of the specifics of the Prague public transport network – day and night lines are not always the same. In this case, the optimal path between the same nodes can be different according to the departure time. To solve it, the application decides what matrix have to be loaded according to departure time.

**Algorithm 1** Path matrices creating

```plaintext
1: function CreatePathMatrix(nodes, isNight)
2:     n ← nodes count
3:     time ← isNight ? 01:00 AM : 10:00 AM
4:     weight ← n × n weight matrix for the current time
5:     path ← n×n path matrix built with Floyd-Warshall algorithm
6:     return path
7: end function
```

### 3.2 Routes Search Algorithm

In this version of the application route representation was an ordered array of departures (entity `Departure`). To find a route, a custom algorithm that worked with Floyd–Warshall matrices (more at section 4.1.1) was used. Pseudocode is presented in algorithm 2.

First, the algorithm loads a path matrix according to `date` and builds a path between `nodeFrom` and `nodeTo` with the loaded matrix. The built path is an array of nodes, nothing more. Then, the algorithm finds appropriate departures from the start of the path and adds those departures to the result.
3. Analysis of the First Version

Algorithm 2 Route searching
1: procedure FindRoute(nodeFrom, nodeTo, date)
2: matrix ← day or night path matrix according to date
3: path ← nodes path from nodeFrom to nodeTo from matrix
4: for i in 1 to path.count do
5: node ← path[i]
6: tmp ← node.departures that have a node from path on a way
7: best ← the best departure from tmp
8: i ← max. number of nodes that can be reached with best
9: date ← best.time + best.minutesInWay + 1 minute
10: add best departure to result
11: end for
12: end procedure

For every line change, one extra minute is added to date to provide time for transfer. If the algorithm must choose between departures (algorithm 2, line 7), it uses a comparison function as presented in algorithm 3. The coefficient 6 was chosen because it produced better results in the tests than other options.

Algorithm 3 Departures comparison
1: function Comparator(firstDeparture, secondDeparture)
2: stop ← difference in the number of stops before transfer
3: time ← difference in the departure time (minutes)
4: return (6 * stop) > time \[6 is a constant coefficient\]
5: end function

3.3 Conclusion

As depicted in table 2.1, the database size is relatively big for the task. Another database structure could exclude a lot of useless information. Moreover, because the question about data structure format is not closed, different formats must be tested to determine which is the most optimal. This can also affect the processing speed of fetch requests.

Algorithmic problems are even more noticeable. After time profiling, it became clear that the hardest part of the algorithm was departure filtering at algorithm 2, line 6. Generally, this part took approximately eighty percent of all computation time. This is a significant amount of time when you consider the average search time of six seconds. For example, another offline application CG Transit from the previous chapter has average search time six hundred milliseconds. Furthermore, considered algorithm often provides non-optimal routes with many transfers. The main reason for this is an incorrect approach
3.3. Conclusion

to the task. The algorithm has no variety when the skeleton of the path is
given. It means, that when the path is built from matrix (algorithm 2 line 3),
nothing can significantly change the constructed path. Despite the fact
that the matrix contains optimal routes, it is composed for a certain time
(algorithm 1 line 3). At other times, the matrix may not provide the most
optimal path.

It is obvious that data structure and route search algorithms should be
replaced by more advantageous options. The list of the most suitable solutions
is presented in the next chapter.
This chapter reviews the most popular existing algorithms for the shortest-path problem and designs a data structure that can be used to improve the application’s performance.

4.1 Routes Search Algorithm

Mobile devices are very sensitive to increased complexity, which is why it is important to revise existing solutions and compare them due to the problem defined in the section 1.2. A lot of works in the field of transportation networks (e.g. [3, 19, 2]) and time-dependent shortest path problem (e.g. [20]) tend to use Dijkstra’s algorithm modifications. Three of Dijkstra’s modifications and a Floyd–Warshall algorithm, used in the first version of the application, are reviewed below.

4.1.1 Floyd–Warshall

In spite of the title of the section, the algorithm considered here is rather a brute–force with using Floyd–Warshall matrices. It is the algorithm used in the first version of the application. It was reviewed in detail in section 3.2 (algorithm 3), where the first version of the application is also revised. The following section describes the basis of the Floyd–Warshall algorithm [18].

The Floyd–Warshall algorithm builds a matrix that represents the shortest path between all pairs of vertices and has time complexity $O(|V|^3)$. Pseudocode is presented in algorithm 4. A list of vertices between the source and target nodes that provide the Floyd–Warshall matrix is not enough to compute a departure path quickly enough. The possible solution to this problem could be storing departures instead of nodes in the matrices, but the algorithm would then need about $7 \times 24 \times 4 = 672$ matrices (one matrix for every 15 minutes in a week) with increased size. That is not possible but would provide a linear complexity $O(n)$, where $n$ is the number of nodes in the path.
Algorithm 4 Floyd–Warshall algorithm

1. procedure FWMatrix(G)
2. matrix[i, j] ← i = j ? 0 : ∞ ▷ ∀i, j ∈ V(G)
3. matrix[i, j] ← E(G)(i, j) ▷ ∀(i, j) ∈ E(G)
4. for i in V(G) do
5.   for j in V(G) do
6.     for k in V(G) do
7.       if matrix[j, k] > matrix[j, i] + matrix[i, k] then
8.         matrix[j, k] ← matrix[j, i] + matrix[i, k]
9.       end if
10.   end for
11. end for
12. end for
13. end procedure

In sum, the Floyd–Warshall algorithm is an effective algorithm for some tasks. But the problem cannot be solved with this algorithm in sufficient time, because the main purpose of the Floyd-Warshall algorithm is to compute a matrix for all pairs of vertices. However, the Prague public transport network contains more than one thousand stops and more than one and a half million departure times that are too much to calculate or store with a mobile device. Moreover, transportation network is a time-dependent graph, so all routes can’t be described with the only one matrix. This algorithm is better suited for static graphs.

4.1.2 Dijkstra

The Dijkstra algorithm [21] is probably the most popular algorithm to address the shortest path problem, despite the fact that it was invented in 1959. It is easy in implementation and provides good time and space complexity. The worst-case performance for the original Dijkstra algorithm is $O(|V|^2)$. In the algorithm, 5 is given a modification of Dijkstra algorithm using a priority queue. This modification can lead to faster computing than original Dijkstra algorithm and has a better worst-case performance $O(|E| + |V| \log |V|)$.

Priority queue is a data type similar to a usual queue, but each element in the priority queue has a “priority” value. The priority queue has three basic operations: extract_min, add_with_priority and decrease_priority. All operations for the Fibonacci priority queue have a constant $O(1)$ time complexity, except for extract_min with a $O(\log n)$ time complexity. Priority queues are reviewed in more detail in [22].

Despite the simplicity, the Dijkstra algorithm has a good chance to produce desirable results. Problems may arise with route search between nodes that are too far, because the basic Dijkstra algorithm is an uninformed algorithm and
4.1. Routes Search Algorithm

Algorithm 5 Dijkstra algorithm with priority queue

1: procedure Dijkstra\((G, source)\)
2:     for \(i\) in \(V(G)\) do
3:         \(\text{dist}[i] \leftarrow i \neq \text{source} \; ? \; \infty : 0\)
4:         queue.add_with_priority\((i, \text{dist}[i])\)
5:     end for
6:     while \(u \leftarrow \text{queue}.\text{extract}\_\text{min}\) do
7:         for all neighbors \(v\) of \(u\) do
8:             if \(\text{dist}[u] + E(G)(u, v) < \text{dist}[v]\) then
9:                 \(\text{dist}[v] \leftarrow \text{dist}[u] + E(G)(u, v)\)
10:                \(\text{prev}[v] \leftarrow u\)
11:               queue.decrease_priority\((v, \text{dist}[u] + E(G)(u, v))\)
12:         end if
13:     end for
14: end while
15: end procedure

therefore its performance will decrease as distance increases between nodes. To find out if these problems will have a critical impact on performance, some tests must be run. Results of these tests can be found in chapter 5.

4.1.3 Bidirectional Dijkstra

The bidirectional Dijkstra algorithm is a modification of the Dijkstra algorithm to include one important feature. The algorithm runs from both source and target vertices at the same time. The path is found when the two meet in the middle. The only thing that must be added to the Dijkstra algorithm to get bidirectional Dijkstra algorithm is vertices marking by both instances.

Figure 4.1: Dijkstra (left) versus Bidirectional Dijkstra (right) dispersion. Red dots represent source and target vertices, the gray circle represents explored vertices, and the green dot represents a common vertex to connect two paths.
4. Design

Theoretically, this modification can solve the problem previously described in section 4.1.2 relating to the Dijkstra algorithm: “...performance will decrease with distance increasing between nodes”. If the search starts from both of the vertices, the dispersion will be significantly less. This is schematically depicted in figure 4.1. But there is a reason why this algorithm can not be used for the problem.

This paragraph is a paraphrase of thesis [21] about bidirectional Dijkstra algorithm. In this thesis it is said that in a public transportation network we cannot execute a search starting at target node because we do not know at what time to begin (a necessary parameter when searching a transit network). We will not know that time until we have executed a forward search and determined at which time we will arrive at target node. Only then can we begin executing the search in the backward direction. This makes a bidirectional Dijkstra search very difficult to implement unless we permit estimation of the arrival time which removes optimality. This is unfortunate because many nice techniques have been developed using bidirectional search.

4.1.4 A*

Another variation of Dijkstra’s algorithm is the A* [3, 20]. It achieves better performance by using heuristic functions [23]. These functions inform search algorithms by choosing the best alternative based on the available information. Specifically, A* algorithm selects alternatives that minimize equation 4.1 where \( n \) is the current vertex in which the algorithm is located, \( g(n) \) is a cost function of the path from start to the current vertex and \( h(n) \) is a heuristic function that approximates cost from the current vertex to the target.

\[
f(n) = g(n) + h(n)\tag{4.1}
\]

The obvious measure of cost is time. Hence, function \( g(n) \) provides the time from the start of the search to arrival to the current node. The heuristic function \( h(n) \) must approximate the time needed to get from the current node to the target. As was mentioned in section 1.3, every stop in GTFS format has the location property, so the fastest way to approximate the needed time is to use distance with a certain coefficient.

The pseudocode is presented in the algorithm 6. The algorithm uses standard implementation with lists, but it can be improved with different techniques. For example, priority queues can be used instead of lists, similar to the Dijkstra algorithm from section 4.1.2. More detailed these options will be considered in the next chapter.

Short analysis of all four algorithms indicates that A* has the greatest chance of success. The main feature is that this algorithm is informed and has good resistance to the problem of performance decreasing when distance increases. Anyway, algorithms will be tested detailed in the next chapter.
Algorithm 6 A* algorithm

1: procedure AStar(G, source, target)
2: open.add(source)
3: g_score, f_score ← map with default value ∞
4: g_score[source] ← 0
5: f_score[source] ← h(source)
6: while open is not empty do
7:   n ← node from open with lowest f_score
8:   if n = target then
9:     break
10: end if
11: open.remove(n)
12: closed.add(n)
13: for all neighbors v of n do
14:   if v ∈ closed then
15:     continue
16: end if
17:   if v ∉ open then
18:     open.add(v)
19:   end if
20:   if g(v) < g_score[v] then
21:     path[v] ← n
22:     g_score[v] ← g(v)
23:     f_score[v] ← g(v) + h(v)
24:   end if
25: end for
26: end while
27: end procedure

4.2 Data Structure

Another problem is the designing of a suitable data structure for all needed data that is parsed from GTFS. The problem is described in section 1.4.

First, the format for stored data must be chosen. There are two options: relational database or raw data. To select the right option, some tests must be processed, the results for which can be found in the next chapter.

Second, the data structure must be defined. The main problem of the database structure from the first version of the application (depicted in figure 3.1) is data repetition. Departures from the same trip have the same Departure.days, Departure.startDate, and Departure.endDate service attributes, because each trip has the only one associated service. It is a mistake to repeat this data for every instance of Departure. The best way to reduce Departure entities is presented in study [3]. Trip entities with
4. Design

Trip.departureTimes arrays of the departure times can be used instead of all Departure entities. Moreover, the entity Stop is not required, because route search algorithm uses data from the Node entity. To completely reduce the Stop entity, Stop.lat and Stop.lon attributes must be moved into the Node entity. The Matrix entity can also be removed if the Floyd–Warshall algorithm will not used.

Figure 4.2: Revised data structure. Relationships mean foreign keys. One arrow means "to-one" and double arrow means "to-many" relationships.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>lat</td>
<td>The latitude of the Node.</td>
</tr>
<tr>
<td></td>
<td>lon</td>
<td>The longitude of the Node.</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>Name of the Node.</td>
</tr>
<tr>
<td></td>
<td>trips</td>
<td>Array of all trips, that go through the Node.</td>
</tr>
<tr>
<td>Trip</td>
<td>departureTimes</td>
<td>Array of all departure times from source Node to the target Node ordered according to Trip.nodes array. Additionally, it has one more element at the last position with arrival time at the target Node.</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>Name of the line to which the trip belongs.</td>
</tr>
<tr>
<td></td>
<td>nodes</td>
<td>Ordered array of nodes, that shows the order in which the trip goes through stops.</td>
</tr>
<tr>
<td></td>
<td>service</td>
<td>Link to the Service entity.</td>
</tr>
<tr>
<td>Service</td>
<td>days</td>
<td>Integer mask, that shows on what days of the week the trip works.</td>
</tr>
<tr>
<td></td>
<td>endDate</td>
<td>Date showing until what day the trip works.</td>
</tr>
<tr>
<td></td>
<td>startDate</td>
<td>Date showing from what day the trip works.</td>
</tr>
<tr>
<td></td>
<td>trips</td>
<td>Array of all trips, that have assigned this service.</td>
</tr>
</tbody>
</table>
4.3 Conclusion

Dijkstra’s algorithm variations can show much better results than algorithm from the first version of the application. After short analysis in this chapter, it can be confidently said that one of the described algorithms will be used as an algorithm for the new version of the application. Anyway, these algorithms are not intended for time-dependent graph, such as transportation network. So, chosen algorithm must be modified to work with Prague public transport network. More about implementation in chapter 5.

Designed data structure is presented in figure 4.2 and described in table 4.1. The main advantages of this structure, in contrast to the previous one, are a reduced number of entities and a significantly smaller size (more in chapter 5). However, this structure also has disadvantages.

First, the algorithm cannot retrieve all departures from the Node, because the Node only has information about trips. This means that the algorithm must search for needed departures in the Trip.departureTimes array according to the Trip.nodes array. However, in the new version of the application it may not be necessary to retrieve all departures. All trips in Node.trips attribute will be ordered by departure time from the node and departures can be extracted up to a certain constant. In this case, this problem will not have a big effect on the performance.

Another disadvantage is that in this implementation, the Trip entity has no attribute with arrival times. It was removed because of specific of Prague public transport network to reduce the size of data structure. The next departure time on the trip is used as an arrival time because it is almost always the same time. However, the departure time from the target node does not exist, which is why the last element added to the Trip.departureTimes array is the arrival time at the target node. Processed tests showed that such data structure helps to significantly reduce the size of the database with almost no effect on the found routes. In the worst case, for a very small number of trips, the arrival time will differ by one minute.
Chapter 5

Implementation

The main aim of the thesis is to implement a new version of the application. The goal of this chapter is to provide the important implementation details and describe designed optimizations.

5.1 Application Structure

The new version of the application, like the old one, is written in Objective-C. Another option was Swift language \textsuperscript{24}, but rewriting the whole application in another language would be complicated. Besides, no one guarantees any increase in performance of an application.

It does not make sense to describe the whole structure of the application, because all iOS applications has almost identical structure. Below are described files, that have impact on the route search algorithm and data structure. Files \texttt{Algorithms.h} and \texttt{Algorithms.m} contain almost everything about route search algorithm and another help functions. File \texttt{AppDelegate.h} contains control macros, e.g. for turning on database loading mode. Transportation network is represented with files \texttt{Database.h}, \texttt{Database.m} and its derivatives \texttt{Database*}. All files have a clear structure and are conveniently placed in the project.

5.2 Route Search Algorithm

All route search algorithms from the previous chapter except for bidirectional Dijkstra can be implemented and compared to choose the best one. As follows from the previous chapter, the A* algorithm has the highest chance of success. Its implementation is shown in algorithm \texttt{A*}. Some changes were applied unlike basic algorithm \texttt{Dijkstra}, because the public transport network is a time-dependent graph and a path is an array of departures, not nodes. That is why the new algorithm stores and compares departures instead of nodes. Furthermore,
the algorithm was implemented using a priority queue. This helps to speed up the algorithm; in algorithm at line 7, acquiring the object with lowest \( f(v) \) has linear time complexity \( O(n) \), but it has constant time complexity \( O(1) \) with priority queue implementation. An open source solution named BEPriorityQueue was used instead of implementing a priority queue from scratch.

Many implementation details can be found with the new algorithm. E.g. at line 9, the algorithm extracts a departure with the lowest priority value and uses it to find an optimal path. The priority value for extracted departure is received to calculate priority values for the newly added departures. Each priority value calculates at lines 28 – 29 and it is a sum of heuristic and cost functions (equation 4.1) for the considered departure created at line 25. The heuristic function is based on the simplest Euclidean distance between the considered node and target node with a constant coefficient of 0.7. The cost function is a number of seconds from the start of searching to the arrival time of the considered departure, multiplied by coefficient 0.5. Additionally, transfer fine is added to priority value to penalize unnecessary line changes. Transfer fine is set to 2000 if the line is changed, to 1000 if one line of metro is changed for another line of metro, and to 0 if there is no transfer. Another coefficient is used at line 17 of the algorithm that limits the number of obtained trips by 40 to prevent the queue from overflowing and to speed up calculating. The probability that the optimal departure will come after another 40 departures is very low, but a lower coefficient has almost no impact on the algorithm’s performance. When the target node is reached, the path is reconstructed from the last departure using the path dictionary. All coefficients used in the application are easily replaceable and are chosen after thorough testing of the optimality of found routes. Most of them can be changed in file AppDelegate.h. For example, constant named C_TIME_COEFFICIENT represents a coefficient used at line 29 (0.5 in the final implementation) and constant named C_DISTANCE_COEFFICIENT is used at line 28 (0.7 in the final implementation).

As was mentioned in the previous chapter, the Dijkstra algorithm is similar to A*. Because A* implementation with priority queue was used, Dijkstra priority queue algorithm 5 can be achieved by removing heuristics from algorithm 7. Another change, such as using departures instead of nodes, must also be applied to make the algorithm work. That is why another pseudocode for Dijkstra algorithm implementation is not provided.

Table 5.1 compares all algorithms on the same routes. All calculations were conducted under the same conditions with the same data structure and content. Exception is the algorithm from the first version of the application, that used Floyd-Warshall matrices to find a route. Results for this algorithm are taken from the first version of the application, because the algorithm needs old data structure. Description of the new data structure, with the help of which another algorithms were tested, is provided in the next section.
Algorithm 7 A* algorithm

1: function AStar(source, target, date)
2:     firstIteration ← true
3:     while !open.isEmpty or firstIteration do
4:         if firstIteration then
5:             value, time, distance ← 0
6:             node ← source
7:             firstIteration ← false
8:         else
9:             departure, value ← open.extract_min
10:            node ← departure.nodeTo
11:            time ← seconds from currentDate to arrival time of departure
12:            distance ← euclidean distance between node and target
13:        end if
14:     if node = target then
15:         return departures array reconstructed with path
16:     end if
17:     trips ← 40 trips from node with the closest departure
18:     for all trip in trips do
19:         if trip.service doesn't work at currentDate then
20:             continue
21:         end if
22:         transferFine ← fine for the transfer
23:         newNode ← next node on the trip's way
24:         newDeparture ← created departure from trip for node
25:         newDistance ← euclidean distance between newNode and target
26:         newTime ← seconds from currentDate to arrival time of newDeparture
27:         newValue ← value + 0.7 · (newDistance − distance) +
28:                 0.5 · (newTime − time) + transferFine
29:         if newDeparture ∉ closed then
30:             open.add_with_priority(newDeparture, newValue)
31:             path[newDeparture] = departure
32:             closed.add(newDeparture)
33:         end if
34:     end for
35:     end while
36: end function
Table 5.1: Comparing algorithms under the same conditions. In the table are given time intervals spent to find a route.

<table>
<thead>
<tr>
<th>Route pair</th>
<th>Optimized brute-force</th>
<th>Dijkstra</th>
<th>A*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dejvická – Můstek</td>
<td>4.28 sec</td>
<td>50 ms</td>
<td>15 ms</td>
</tr>
<tr>
<td>Marjánka – Anděl</td>
<td>8.85 sec</td>
<td>17.35 sec</td>
<td>57 ms</td>
</tr>
<tr>
<td>Hulínček – Hasova</td>
<td>7.98 sec</td>
<td>&gt; 300 sec</td>
<td>140 ms</td>
</tr>
<tr>
<td>Hládkov – Volha</td>
<td>13.78 sec</td>
<td>&gt; 300 sec</td>
<td>192 ms</td>
</tr>
<tr>
<td>Stražní – I. P. Pavlova</td>
<td>4.97 sec</td>
<td>90 ms</td>
<td>133 ms</td>
</tr>
<tr>
<td>Kublov – Vozovna Zížkov</td>
<td>10.79 sec</td>
<td>&gt; 300 sec</td>
<td>70 ms</td>
</tr>
<tr>
<td>Dívčí hrady – Kajetánka</td>
<td>6.92 sec</td>
<td>&gt; 300 sec</td>
<td>1.29 sec</td>
</tr>
<tr>
<td>Trojská – Motol</td>
<td>4.36 sec</td>
<td>&gt; 300 sec</td>
<td>105 ms</td>
</tr>
<tr>
<td>Bílá Hora – Bílá labuť</td>
<td>3.70 sec</td>
<td>11.83 sec</td>
<td>2.09 sec</td>
</tr>
<tr>
<td>Palmovka – Lhotka</td>
<td>5.89 sec</td>
<td>&gt; 300 sec</td>
<td>140 ms</td>
</tr>
<tr>
<td>Terminál 3 – Hotel Golf</td>
<td>5.30 sec</td>
<td>&gt; 300 sec</td>
<td>4.85 sec</td>
</tr>
<tr>
<td>Zlíčín – Bílá Hora</td>
<td>3.95 sec</td>
<td>&gt; 300 sec</td>
<td>113 ms</td>
</tr>
<tr>
<td>Lotyšská – Kněžská luka</td>
<td>3.75 sec</td>
<td>21.89 sec</td>
<td>854 ms</td>
</tr>
<tr>
<td>U Památníku – Ruská</td>
<td>1.14 sec</td>
<td>1.83 sec</td>
<td>828 ms</td>
</tr>
</tbody>
</table>

Table 5.1 illustrates that the A* algorithm has better results than other options; the algorithm was accordingly chosen for the new version of the application. However, performance decreases could be noted for routes “Dívčí hrady – Kajetánka”, “Bílá Hora – Bílá labuť” and “Terminál 3 – Hotel Golf”. The reason for this is small implementation issues. Despite this, almost all routes can be found in a fraction of a second. The Dijkstra algorithm is an uninformed algorithm, which poses a problem previously described in section 4.1.2 that relates to performance decreasing with increases in distance.

5.3 Data Structure

It was decided that the new version of the application should use the GTFS file `agency.txt`. This file contains information about transit agencies that provided data for this feed. Data from this file was used to filter out lines provided from agencies other than PPTC. It helped to reduce database size and is not affected by route search in Prague because most of the other agencies provided lines for suburb areas.

One more optimization was applied to separate lines that were not intended for ordinary movement in Prague. Within this optimization were separated lines with identification numbers in the range 251 – 899. According to PPTC [20], these lines were devised for school buses and regional routes.

A lot of suburban nodes had zero trips after these optimizations were applied. Nodes with empty departures array do not do any good and complicate nodes search process, that’s why was decided to remove these nodes from the
In addition to route search from node $A$ to node $B$ at the current time, the new version of the application has implemented some user enhancements. They
5. **Implementation**

are briefly described in this section.

- **Departure time**
  
  Users can explicitly set departure time and day (only today or tomorrow) from which to search a route. There is no additional algorithm for this; another date is simply passed to algorithm as a *date* argument.

- **Next route**
  
  After a route is found, users can search for the next one by pressing the appropriate button. This feature uses the same algorithm for route search, but as a *date* argument is sent the first departure time of the found route with five extra minutes. This ensures that a new route will be different with a later departure time.

- **Via**
  
  Users can specify an intermediate node C for the route search from A to B. First, the algorithm finds a route between A and C. Then, it searches a route from C to B at arrival time to C. After both routes are found, the algorithm unite them into one.

- **Locating**
  
  The fastest way to specify nodes is to use locating. This feature compares Euclidean distance from the device’s location to all nodes and chooses the closest. The nodes location is also used in heuristic functions, so this enhancement does not require any additional data and has no impact on the data structure. This function is very convenient for the specification of departure node.

5.5 **Distribution**

The new version of the application, like the first one, will distribute with Apple’s official App Store. However, an application needs to be reviewed by Apple’s employees before publishing. Unfortunately, at the time of writing, the application is not available for downloading from App Store, but it will as soon as possible.

5.6 **Conclusion**

All implemented improvements, including the new algorithm and data structure, showed a significant increase in productivity. The basic A* algorithm was successfully modified for using in time-dependent transportation networks.
The data structure size has been reduced more than twice. Anyway, implementation contains some irregularities that must be fixed in the next version of the application.

Despite the fact that the application is not yet available on the App Store, internal testing was processed to get feedback from users. All testers noted much faster route search and absence of unnecessary transfers. Some of them paid attention to the fact that the new version of the application often prefers land transport (buses, trams) instead of the metro. But everyone agreed that the new version is much better than the old one.
Table 6.1 was compiled to illustrate how the new version of the application works in the real life. Some popular routes were chosen to provide time spent on the search with the old and the new algorithms.

Table 6.1: Comparing the first version and the new version of DPP Offline under the same conditions. In the table is given time spent to find a route.

<table>
<thead>
<tr>
<th>Route</th>
<th>The first version</th>
<th>The new version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marjánka – Anděl</td>
<td>8,85 sec</td>
<td>57 ms</td>
</tr>
<tr>
<td>Dejvická – Můstek</td>
<td>4,28 sec</td>
<td>15 ms</td>
</tr>
<tr>
<td>Husinecká – Hasova</td>
<td>7,98 sec</td>
<td>140 ms</td>
</tr>
<tr>
<td>Hládkov – Volha</td>
<td>13,78 sec</td>
<td>192 ms</td>
</tr>
<tr>
<td>Strážní – I. P. Pavlova</td>
<td>4,97 sec</td>
<td>133 ms</td>
</tr>
<tr>
<td>Kublov – Vozovna Žižkov</td>
<td>10,79 sec</td>
<td>70 ms</td>
</tr>
<tr>
<td>Dívčí hrady – Kajetánka</td>
<td>6,92 sec</td>
<td>1,29 sec</td>
</tr>
<tr>
<td>Trojští – Motol</td>
<td>4,36 sec</td>
<td>105 ms</td>
</tr>
<tr>
<td>Na Smetance – Vítězné nám.</td>
<td>4,67 sec</td>
<td>60 ms</td>
</tr>
<tr>
<td>Bílá Hora – Bílá labuť</td>
<td>3,70 sec</td>
<td>2,09 sec</td>
</tr>
<tr>
<td>Palmovka – Lhotka</td>
<td>5,89 sec</td>
<td>140 ms</td>
</tr>
<tr>
<td>Terminál 3 – Hotel Golf</td>
<td>5,30 sec</td>
<td>4,85 sec</td>
</tr>
<tr>
<td>Spojovací – Čechovo nám.</td>
<td>3,92 sec</td>
<td>480 ms</td>
</tr>
<tr>
<td>Zličín – Bílá Hora</td>
<td>3,95 sec</td>
<td>113 ms</td>
</tr>
<tr>
<td>Masarykovo nádr. – Opatov</td>
<td>2,34 sec</td>
<td>90 ms</td>
</tr>
<tr>
<td>Lotyšská – Kněžská luka</td>
<td>3,75 sec</td>
<td>854 ms</td>
</tr>
<tr>
<td>U Památníku – Ruská</td>
<td>1,14 sec</td>
<td>828 ms</td>
</tr>
<tr>
<td>Albertov – Apolinářská</td>
<td>&gt; 300 sec</td>
<td>&gt; 300 sec</td>
</tr>
<tr>
<td>Stadion Strahov – Limuzská</td>
<td>3,62 sec</td>
<td>162 ms</td>
</tr>
</tbody>
</table>
6. Testing

Problem was found for the route "Albertov – Apolinářská". When the route does not exist, the algorithm slips into an infinite loop until user stops the search. Moreover, the search can be very long when user by day searches the route between nodes, departures between which work only at night. However, this situation is quite rare for the Prague public transport network.

Optimality of the found routes is subjective. However, it can be said that the situation has improved significantly compared to the previous version of the application. Some examples are depicted in figures 6.1, 6.2, 6.3 and 6.4.

Figure 6.1: Route search comparing. The first version (left) versus the new version (right). Route Kublov – Vozovna Žižkov.

Figure 6.2: Route search comparing. The first version (left) versus the new version (right). Route Dívčí hrady – Kajetánka.
Figure 6.3: Route search comparing. The first version (left) versus the new version (right). Route Terminál 3 – Hotel Golf. 

![Route comparison 1](image1)

Figure 6.4: Route search comparing. The first version (left) versus the new version (right). Route Zlčín – Bílá Hora. 

![Route comparison 2](image2)

In figure 6.4, the first version offers to use a bus number 380, but database from the new version doesn’t contain information about inter-urban buses to not confuse users and reduce database size.

All tests here and from previous chapters were performed by the Apple iPhone SE. It has dual-core A9 processor with clock rate 1.8 GHz [28].
Conclusion

The main aim of the thesis was to design and implement a new version of the application with an improved search time that finds more optimal routes. As follows from the previous chapter, the goal was definitely reached. Performance of the new version has gone up significantly, and the provided routes have no unnecessary transfers. The data structure was optimized and its size was significantly decreased. However, the application has some little implementation issues that must be fixed in the next version. For example, handling the situation when the route between two nodes does not exist. Anyway, the new version of DPP Offline, developed within this thesis, puts the application on par with the best applications for offline route search.

Future Improvements

Additional functions can be added to the application. Some of them are proposed below.

- Foot edges

  Foot edges can save a lot of time for the user. Occasionally, it will be faster to walk for a distance than to wait for the next departure. Distance between nodes can be used to approximate walk time. Many existing applications have already implemented this feature, but it can be expanded with a customizable maximum distance that the user is ready to walk.

- Effective route planning with GPS data

  If the user allows the application to use GPS data, it can be used for effective route planning. The idea is to track user’s location and dynamically change the route when the user has no time to catch the needed trip.
• Show many routes at once

This feature belongs to the UI improvements. It is already implemented in applications described in chapter 3. It can be seen in figures 2.1, 2.2 and 2.3. It is more convenient to have several options at once, but this feature can decrease performance because several routes must be found simultaneously.
Bibliography


Acronyms

CTU  Czech Technical University
FIT  Faculty of Information Technology
iOS  iPhone Operating System
GTFS  General Transit Feed Specification
CSV  Comma-Separated Values
PPTC  Prague Public Transit Company
UML  Unified Modeling Language
GPS  Global Positioning System
Appendix B

Contents of enclosed media

- readme.txt .................. short description of CD’s content
- thesis.pdf .................... this thesis in PDF format
- src
  - DPP Offline .................. source code of DPP Offline
  - thesis ...................... source code of this thesis in \LaTeX\ format
  - jrddata.zip .............. original GTFS data from PPTC for 2018-05-14