FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

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.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 6, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

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

In Prague on May 14, 2018 …………………

Citation of this thesis

Kandaurov, Artem. *Mobile Application for Route Search in Prague Public Transport.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

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

6 Testing **33**

Conclusion **37**
Future Improvements 37

Bibliography **39**

A Acronyms **43**

B Contents of enclosed media **45**

List of Figures

2.1 User Interface of IDOS. 7
2.2 User Interface of PID Info. 8
2.3 User Interface of CG Transit. 8
2.4 User Interface of DPP Offline. 9

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. 11

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. 19
4.2 Revised data structure. Relationships mean foreign keys. One arrow means ”to-one” and double arrow means ”to-many” relationships. 22

6.1 Route search comparing. The first version (left) versus the new version (right). Route Kublov – Vozovna Žižkov. 34
6.2 Route search comparing. The first version (left) versus the new version (right). Route Dívčí hrady – Kajetánka. 34
6.3 Route search comparing. The first version (left) versus the new version (right). Route Terminál 3 – Hotel Golf. 35
6.4 Route search comparing. The first version (left) versus the new version (right). Route Zličín – Bílá Hora. 35

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 version 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 [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* [2] is a graph that utilizes a cost function $cost(e, t)$ 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, 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 [6].

Table 1.1: Required GTFS files.

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

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.

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

More information about the GTFS format can be found e.g. in the FIT CTU master's thesis [7].

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.

CHAPTER 2

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 [8] for the iOS platform.

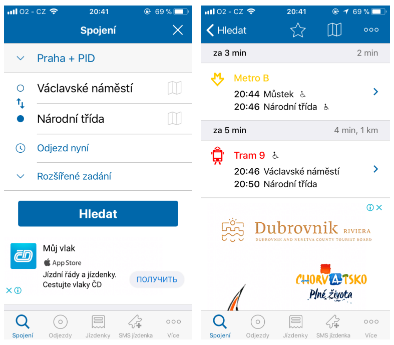
2.1 Existing Software Products

All statistical data in this section is collected by the Apptrace [9] service. The information presented here is correct as of 30th April 2018.

- IDOS [10]

  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.

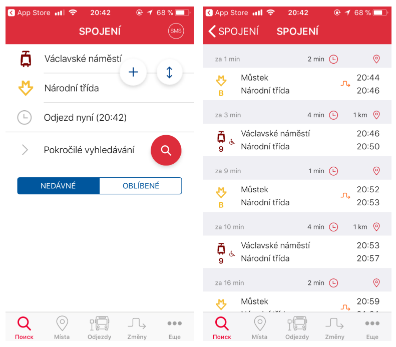
Figure 2.1: User Interface of IDOS.



- PID Info [11]

  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.

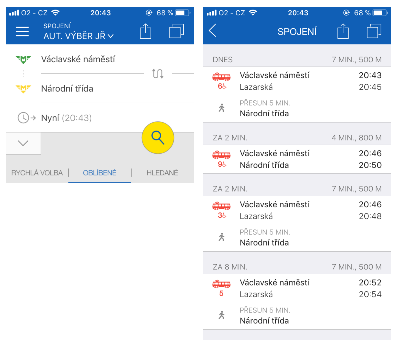

App Store 20:42 68 %
SPOJENÍ
SMS
Václavské náměstí
Národní třída
Odjezd nyní (20:42)
Pokročilé vyhledávání
NEDÁVNÉ OBLÍBENÉ
Поиск Místa Odjezdy Změny Еще

App Store 20:42 68 %
SPOJENÍ SPOJENÍ
za 1 min 2 min
B Můstek 20:44
Národní třída 20:46
za 3 min 4 min 1 km
9 Václavské náměstí 20:46
Národní třída 20:50
za 9 min 1 min
B Můstek 20:52
Národní třída 20:53
za 10 min 4 min 1 km
9 Václavské náměstí 20:53
Národní třída 20:57
za 16 min 2 min
Můstek 20:59
Поиск Místa Odjezdy Změny Еще

- CG Transit [12]

  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 [13]. 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.

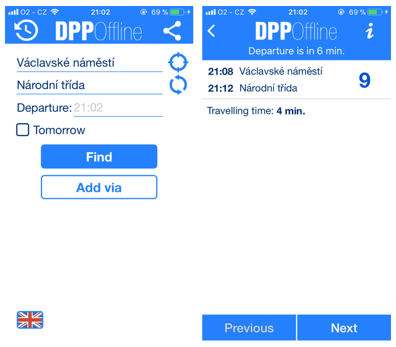

O2 - CZ 20:43 68 %
SPOJENÍ
AUT. VÝBĚR JŘ
Václavské náměstí
Národní třída
Nyní (20:43)
RYCHLÁ VOLBA | OBLÍBENÉ | HLEDANÉ

O2 - CZ 20:43 68 %
SPOJENÍ
DNES 7 MIN., 500 M
6 Václavské náměstí 20:43
Lazarská 20:45
PŘESUN 5 MIN.
Národní třída
ZA 2 MIN. 4 MIN., 800 M
9 Václavské náměstí 20:46
Národní třída 20:50
ZA 2 MIN. 7 MIN., 500 M
3 Václavské náměstí 20:46
Lazarská 20:48
PŘESUN 5 MIN.
Národní třída
ZA 8 MIN. 7 MIN., 500 M
5 Václavské náměstí 20:52
Lazarská 20:54
PŘESUN 5 MIN.
Národní třída

- 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.

	IDOS	PID Info	CG Transit	DPP Offline
Free of charge	Yes	Yes	No	Yes
Offline mode	No	No	Yes	Yes
Other cities	Yes	No	Yes	No
Internal map	Yes	Yes	Yes	No
Contains advertising	Yes	No	No	No
Average search time	800 ms	1 sec	600 ms	6 sec
Database size	–	–	11 MB	205 MB

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.

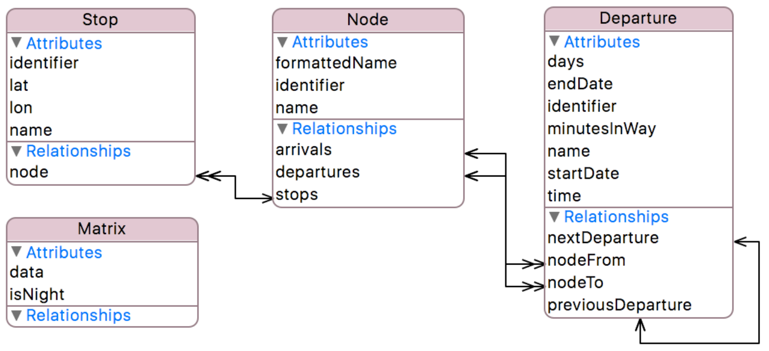


Table 3.1: Description of data structure of the database used in the first version of DPP Offline.

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

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 in the Prague transportation network.

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 $\mathcal{O}(|V|^3)$ [18] 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

```
1: function CreatePathMatrix($nodes, isNight$)
2:     $n \leftarrow nodes$ count
3:     $time \leftarrow isNight$ ? 01:00 AM : 10:00 AM
4:     $weight \leftarrow n \times n$ weight matrix for the current $time$
5:     $path \leftarrow n \times n$ 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.

Algorithm 2 Route searching

```
1: procedure FindRoute($nodeFrom, nodeTo, date$)
2:     $matrix \leftarrow$ day or night path matrix according to $date$
3:     $path \leftarrow$ nodes path from $nodeFrom$ to $nodeTo$ from $matrix$
4:     for $i$ in 1 to $path.count$ do
5:         $node \leftarrow path[i]$
6:         $tmp \leftarrow node.departures$ that have a node from $path$ on a way
7:         $best \leftarrow$ the best departure from $tmp$
8:         $i \leftarrow$ max. number of nodes that can be reached with $best$
9:         $date \leftarrow best.time + best.minutesInWay + 1$ 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 Deaprtures comporison

```
1: function Comparator($firstDeparture, secondDeparture$)
2:     $stop \leftarrow$ difference in the number of stops before transfer
3:     $time \leftarrow$ 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 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.

Chapter 4

Design

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 2), 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 $\mathcal{O}(|V|^3)$. 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 $7 \cdot 24 \cdot 4 = 672$ matrices (one matrix for every 15 minutes in a week) with increased size. That is not possible but would provide a linear complexity $\mathcal{O}(n)$, where n is the number of nodes in the path.

Algorithm 4 Floyd–Warshall algorithm

```
procedure FWMatrix($G$)
    $matrix[i,j] \leftarrow i$ = $j$ ? 0 : $\infty$                ▷ $\forall i, j \in V(G)$
    $matrix[i,j] \leftarrow E(G)(i,j)$                              ▷ $\forall (i,j) \in E(G)$
    for $i$ in $V(G)$ do
        for $j$ in $V(G)$ do
            for $k$ in $V(G)$ do
                if $matrix[j,k]$ > $matrix[j,i]$ + $matrix[i,k]$ then
                    $matrix[j,k] \leftarrow matrix[j,i]$ + $matrix[i,k]$
                end if
            end for
        end for
    end for
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 $\mathcal{O}(|V|^2)$. 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 $\mathcal{O}(|E| + |V| \log |V|)$.

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 $\mathcal{O}(1)$ time complexity, except for `extract_min` with a $\mathcal{O}(\log n)$ 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

Algorithm 5 Dijkstra algorithm with priority queue

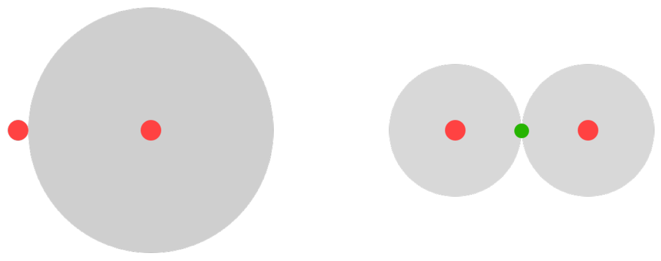
```
procedure DIJKSTRA($G, source$)
    for $i$ in $V(G)$ do
        $dist[i] \leftarrow i \neq source$ ? $\infty$ : 0
        $queue$.add_with_priority($i, dist[i]$)
    end for
    while $u \leftarrow queue$.extract_min do
        for all neighbors $v$ of $u$ do
            if $dist[u]$ + $E(G)(u, v)$ < $dist[v]$ then
                $dist[v] \leftarrow dist[u]$ + $E(G)(u, v)$
                $prev[v] \leftarrow u$
                $queue$.decrease_priority($v, dist[u]$ + $E(G)(u, v)$)
            end if
        end for
    end while
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 [2] 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 5 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.



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 [2] 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:     $g\_score, f\_score \leftarrow$ map with default value $\infty$
4:     $g\_score[source] \leftarrow$ 0
5:     $f\_score[source] \leftarrow h(source)$
6:     while $open$ is not empty do
7:         $n \leftarrow$ 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 $v \in closed$ then
15:                continue
16:            end if
17:            if $v \notin open$ then
18:                $open$.add($v$)
19:            end if
20:            if $g(v) < g\_score[v]$ then
21:                $path[v] \leftarrow n$
22:                $g\_score[v] \leftarrow g(v)$
23:                $f\_score[v] \leftarrow g(v) + h(v)$
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

`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.

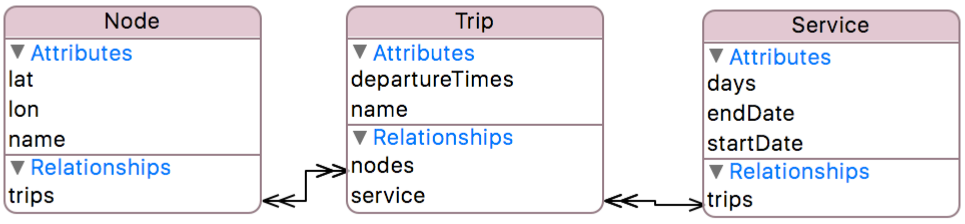


Table 4.1: Description of revised data structure.

Entity	Attribute	Description
Node	`lat`	The latitude of the `Node`.
	`lon`	The longitude of the `Node`.
	`name`	Name of the `Node`.
	`trips`	Array of all trips, that go through the `Node`.
Trip	`departureTimes`	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`.
	`name`	Name of the line to which the trip belongs.
	`nodes`	Ordered array of nodes, that shows the order in which the trip goes through stops.
	`service`	Link to the `Service` entity.
Service	`days`	Integer mask, that shows on what days of the week the trip works.
	`endDate`	Date showing until what day the trip works.
	`startDate`	Date showing from what day the trip works.
	`trips`	Array of all trips, that have assigned this service.

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 [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 `Algorithms.h` and `Algorithms.m` contain almost everything about route search algorithm and another help functions. File `AppDelegate.h` contains control macros, e.g. for turning on database loading mode. Transportation network is represented with files `Database.h`, `Database.m` and its derivatives `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 7. Some changes were applied unlike basic algorithm 6, 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 6 at line 7, acquiring the object with lowest $f(v)$ has linear time complexity $\mathcal{O}(n)$, but it has constant time complexity $\mathcal{O}(1)$ with priority queue implementation. An open source solution named BEPriorityQueue [25] was used instead of implementing a priority queue from a 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 are left 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

```
function AStar($source, target, date$)
    $firstIteration \leftarrow$ true
    while !$open$.isEmpty or $firstIteration$ do
        if $firstIteration$ then
            $value, time, distance \leftarrow$ 0
            $node \leftarrow source$
            $firstIteration \leftarrow$ false
        else
            $departure, value \leftarrow open$.extract_min
            $node \leftarrow departure$.nodeTo
            $time \leftarrow$ seconds from $currentDate$ to arrival time of $departure$
            $distance \leftarrow$ euclidean distance between $node$ and $target$
        end if
        if $node = target$ then
            return departures array reconstructed with $path$
        end if
        $trips \leftarrow$ 40 trips from $node$ with the closest departure
         time to the $currentDate$
        for all $trip$ in $trips$ do
            if $trip$.service doesn't work at $currentDate$ then
                continue
            end if
            $transferFine \leftarrow$ fine for the transfer
            $newNode \leftarrow$ next node on the trip's way
            $newDeparture \leftarrow$ created departure from $trip$ for $node$
            $newDistance \leftarrow$ euclidean distance between $newNode$ and $target$
            $newTime \leftarrow$ seconds from $currentDate$ to arrival time of $newDeparture$
            $newValue \leftarrow value + 0,7 \cdot (newDistance - distance) +$
             $0,5 \cdot (newTime - time) + transferFine$
            if $newDeparture \notin closed$ then
                $open$.add_with_priority($newDeparture, newValue$)
                $path[newDeparture] = departure$
                $closed$.add($newDeparture$)
            end if
        end for
    end while
end function
```

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

	Optimized brute-force	Dijkstra	A*
Dejvická – Můstek	4,28 sec	50 ms	15 ms
Marjánka – Anděl	8,85 sec	17,35 sec	57 ms
Husinecká – Hasova	7,98 sec	> 300 sec	140 ms
Hládkov – Volha	13,78 sec	> 300 sec	192 ms
Strážní – I. P. Pavlova	4,97 sec	90 ms	133 ms
Kublov – Vozovna Žižkov	10,79 sec	> 300 sec	70 ms
Dívčí hrady – Kajetánka	6,92 sec	> 300 sec	1,29 sec
Trojská – Motol	4,36 sec	> 300 sec	105 ms
Bílá Hora – Bílá labuť	3,70 sec	11,83 sec	2,09 sec
Palmovka – Lhotka	5,89 sec	> 300 sec	140 ms
Terminál 3 – Hotel Golf	5,30 sec	> 300 sec	4,85 sec
Zličín – Bílá Hora	3,95 sec	> 300 sec	113 ms
Lotyšská – Kněžská luka	3,75 sec	21,89 sec	854 ms
U Památníku – Ruská	1,14 sec	1,83 sec	828 ms

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 [26], 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 database. The amount of nodes was decreased from 1784 to 1161 as a result. However, this did not have a big impact on the database size.

Filtering a number of trips from the given time occurs at line 17 in the new algorithm 7. The array `node.trips` must be sorted to perform this operation more quickly. Ordering is done in GTFS parsing phase.

As was mentioned in section 4.2, two of the most common formats can be used for storing such data structure. The first option is a relational database. Usually, as a relational database format for iOS is used SQLite. This option was used in the first version of the application. The advantages are simple compatibility with iOS Core Data and a time tested system for database management. However, disadvantages for relational databases also exist, mainly sluggishness, large amounts of metadata, and a long preprocessing time. Another option is using coded raw data with its own structure. The disadvantages of this method are an inability to use links inside the data structure and the absence of any embedded optimizations and infrastructure in general. In return, the structure becomes fully customizable with no excess data.

Creating an SQLite database is a known process which does not make sense to describe, but there are infinite ways to create raw data. The creation of raw data files for this thesis used two classes of Apple's framework Foundation [27]: NSCoder and NSData. NSCoder allows for the conversion of any instance of any class to an NSData object and back. An instance of NSData class is a static byte buffer with an interface for managing this data. Fortunately, the interface of NSData provides methods for its writing and reading to and from a file. The only thing to do is to create class representing data structure showed in figure 4.2 and to fill it with parsed and processed GTFS data. However, this class must contain integer values for referencing to another instance instead of usual links.

After testing both formats, it was decided to use the raw data format. The first reason for this was the result database size. It was circa 180 MB for SQLite format and 75 MB for raw data. This difference can be explained by the large amount of metadata and the need to reverse links for every relationship in Core Data. The second reason, correlated with the first one, is preprocessing time. The device spends approximately thirteen seconds to load the database from a file in SQLite format after the application is started. Using the raw data format, this time decreases to 5 seconds. This time is decreased to approximately three seconds if the raw data file is separated into three different parts (nodes, trips, and services) and the parts are loaded in parallel.

5.4 Functions

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 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 7 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 7 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.

Chapter 6

Testing

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.

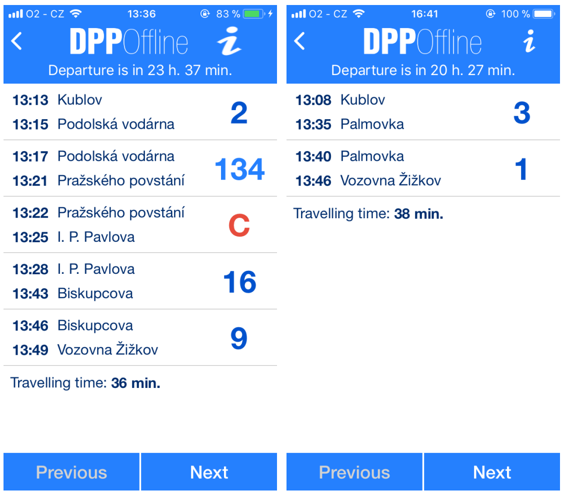
Table 6.1: Comparing the first version and the new version of DPP Offline under the same conditions. In the table is given time spent to find a route.

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

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.



Departure is in 23 h. 37 min.

13:13 Kublov
13:15 Podolská vodárna
2

13:17 Podolská vodárna
13:21 Pražského povstání
134

13:22 Pražského povstání
13:25 I. P. Pavlova
C

13:28 I. P. Pavlova
13:43 Biskupcova
16

13:46 Biskupcova
13:49 Vozovna Žižkov
9

Travelling time: **36 min.**

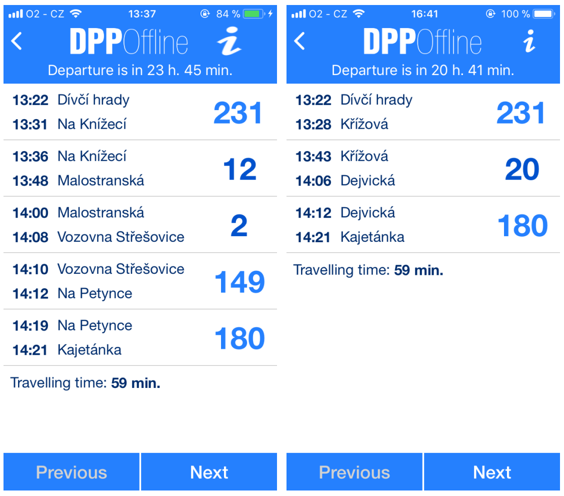
Departure is in 20 h. 27 min.

13:08 Kublov
13:35 Palmovka
3

13:40 Palmovka
13:46 Vozovna Žižkov
1

Travelling time: **38 min.**

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



Departure is in 23 h. 45 min.

13:22 Dívčí hrady
13:31 Na Knížecí
231

13:36 Na Knížecí
13:48 Malostranská
12

14:00 Malostranská
14:08 Vozovna Střešovice
2

14:10 Vozovna Střešovice
14:12 Na Petynce
149

14:19 Na Petynce
14:21 Kajetánka
180

Travelling time: **59 min.**

Departure is in 20 h. 41 min.

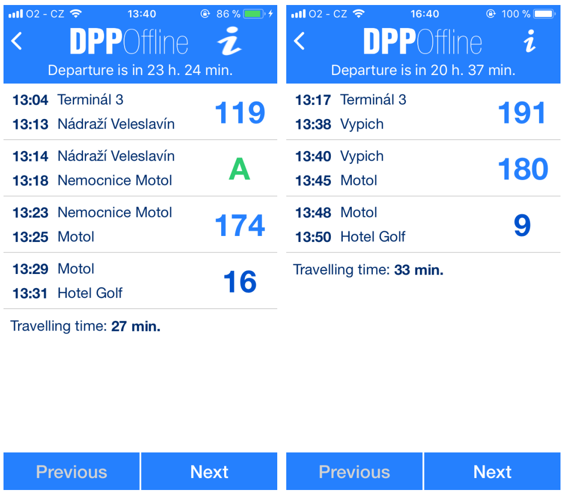
13:22 Dívčí hrady
13:28 Křížová
231

13:43 Křížová
14:06 Dejvická
20

14:12 Dejvická
14:21 Kajetánka
180

Travelling time: **59 min.**

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



Departure is in 23 h. 24 min.

13:04 Terminál 3 13:13 Nádraží Veleslavín	119
13:14 Nádraží Veleslavín 13:18 Nemocnice Motol	A
13:23 Nemocnice Motol 13:25 Motol	174
13:29 Motol 13:31 Hotel Golf	16

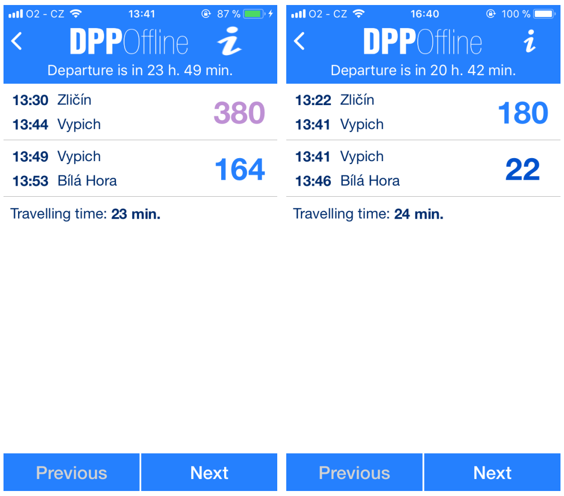
Travelling time: **27 min.**

Departure is in 20 h. 37 min.

13:17 Terminál 3 13:38 Vypich	191
13:40 Vypich 13:45 Motol	180
13:48 Motol 13:50 Hotel Golf	9

Travelling time: **33 min.**

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



Departure is in 23 h. 49 min.

13:30 Zličín 13:44 Vypich	380
13:49 Vypich 13:53 Bílá Hora	164

Travelling time: **23 min.**

Departure is in 20 h. 42 min.

13:22 Zličín 13:41 Vypich	180
13:41 Vypich 13:46 Bílá Hora	22

Travelling time: **24 min.**

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

[1] Mareš, M.; Valla, T. *Průvodce labyrintem algoritmů.* CZ. NIC, zspo, 2017.

[2] Merrifield, T. *Heuristic Route Search in Public Transportation Networks.* Dissertation thesis, University of Illinois at Chicago, 2010.

[3] Szincsák, T.; Vágner, A. Public transit schedule and route planner application for mobile devices. In *Proceedings of the 9th International Conference on Applied Informatics, Eger*, volume 2, 2014, pp. 153–161.

[4] Ahuja, R. K.; Mehlhorn, K.; et al. Faster algorithms for the shortest path problem. *Journal of the ACM (JACM)*, volume 37, no. 2, 1990: pp. 213–223.

[5] Dopravní podnik hl. m. Prahy Jízdní řády GTFS. [online], [cit. 2017-11-28]. Available from: `http://opendata.praha.eu/dataset/dpp-jizdni-rady`

[6] GTFS Reference. [online], [cit. 2018-03-06]. Available from: `https://developers.google.com/transit/gtfs/reference`

[7] Jelínek, R. *Prohledávání dopravních síti v GTFS formátu.* Master's thesis, České vysoké učení technické v Praze, 2017.

[8] App Store – Apple. [online], [cit. 2017-12-12]. Available from: `https://www.apple.com/ios/app-store`

[9] App statistics | App store intelligence | apptrace. [online], [cit. 2017-12-12]. Available from: `http://www.apptrace.com`

[10] MAFRA, a.s. Jízdní řády IDOS. [software], 2011, version 2.8.0 [cit. 2018-05-13]. Available from: `https://itunes.apple.com/cz/app/id473503749`

[11] Regionální organizátor Pražské integrované dopravy. PID Info. [software], 2016, version 1.1.3 [cit. 2018-05-13]. Available from: `https://itunes.apple.com/cz/app/id983071129`

[12] Circlegate. Jízdní řády – CG Transit. [software], 2011, version 4.1.1 [cit. 2018-05-13]. Available from: `https://itunes.apple.com/cz/app/id430848814`

[13] Chaps – SOFTWAROVÁ ŘEŠENÍ PRO OSOBNÍ DOPRAVU. [online], [cit. 2017-12-12]. Available from: `https://www.chaps.cz`

[14] Google Maps. [online], [cit. 2018-05-13]. Available from: `http://maps.google.com`

[15] Kochan, S. G. *Programming in objective-C*. Addison-Wesley Professional, 2011.

[16] SQLite Home Page. [online], [cit. 2017-11-28]. Available from: `https://www.sqlite.org`

[17] Core Data | Apple Developer Documentation. [online], [cit. 2017-11-28]. Available from: `https://developer.apple.com/documentation/coredata`

[18] Hougardy, S. The Floyd–Warshall algorithm on graphs with negative cycles. *Information Processing Letters*, volume 110, no. 8-9, 2010: pp. 279–281.

[19] Martınek, V.; Zemlicka, M. Speeding up shortest path search in public transport networks. DATESO, 2009.

[20] Zhao, L.; Ohshima, T.; et al. A* Algorithm for the time-dependent shortest path problem. In *WAAC08: The 11th Japan-Korea Joint Workshop on Algorithms and Computation*, 2008.

[21] Skiena, S. Dijkstra's algorithm. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, 1990: pp. 225–227.

[22] Cormen, T. H. *Introduction to algorithms*. MIT press, 2009.

[23] Pearl, J. Heuristics: Intelligent search strategies for computer problem solving.

[24] Lardinois, F. Apple Launches Swift, A New Programming Language For Writing iOS And OS X Apps. 2017.

[25] BEPriorityQueue – Objective-c Priority Queue. [online], [cit. 2018-05-04]. Available from: `https://stackoverflow.com/a/37209350/4545903`

[26] JR Portál – Přehled linek. [online], [cit. 2018-05-07]. Available from: `http://jrportal.dpp.cz/jrportal/LineList.aspx?mi=6&t=4`

[27] Foundation | Apple Developer Documentation. [online], [cit. 2018-05-04]. Available from: `https://developer.apple.com/documentation/foundation`

[28] Revealed: iPhone 6S uses 1.8GHz dual-core A9 chip | Trusted Reviews. [online], [cit. 2018-04-22]. Available from: `http://www.trustedreviews.com/news/revealed-iphone-6s-uses-1-8ghz-dual-core-a9-chip-2927237`

Appendix A
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
  - `jrdata.zip` .... original GTFS data from PPTC for 2018-05-14