Proceedings of the 9th International Conference on Applied Informatics
Eger, Hungary, January 29–February 1, 2014. Vol. 2. pp. 153–161
doi: 10.14794/ICAI.9.2014.2.153

Public transit schedule and route planner application for mobile devices*

Tamás Szincsák, Anikó Vágner

University of Debrecen, Faculty of Informatics
sztamas89@gmail.com, vagner.aniko@inf.unideb.hu

Abstract

In this paper we introduce a public transit schedule and route planner application for urban public transportation. The application was developed for the Android Operating System. It works offline and does not need a permanent internet connection.

While transport agencies usually make their schedules available online, browsing them outdoors is not always possible, or requires too much effort on the small screen of a mobile phone. The solutions of this problem are the GTFS databases, which describe the urban public transportation systems. Developers can create applications which use the GTFS databases. However, these databases are not applicable for fast processing needed for route planning on mobile devices with limited resources. Therefore we created a data structure from the GTFS database which is easier to manage, smaller and faster to process. The Android application uses this data structure.

The application lists the departure times, journey times and the stops of the lines. It can display the lines on the map. It also shows the stops nearby with the name of the lines passing through. The user can choose one or more stops and view details about the lines passing through.

The main function of the application is the route planner. It plans at least one route from a starting-point to a destination. Before the route planning the user can set parameters which are taken into account by the route planner.

Keywords: mobile application, public transit, GTFS database, route planner

1. Introduction

Nowadays, many people use smart mobile devices, thus the demand for applications running on those devices is increasing. People need information systems which help their everyday life. There are a lot of well-known desktop applications for

*The publication was supported by the TÁMOP-4.2.2.C-11/1/KONV-2012-0001 project. The project has been supported by the European Union, co-financed by the European Social Fund.

viewing public transit schedules, planning between two points, showing routes on the map, etc. One of the examples is `www.mav-start.hu`, which shows timetable information, plans routes, and gives information about the current positions of trains. [1]. The other example is Google Maps [2], which shows the current position of the user and plans routes for travelling by car, on foot and in many cases by public transport. Other well-known route planner for public transport of Budapest is the route planner of Budapest Transport Plc (BKV) [3].

Users can access these applications using web browsers. In this way they can be used on mobile devices. However, not everyone has a mobile data connection, or they can't access all information they need during travelling.

In this article we introduce a public transit schedule and route planner application for urban public transportation. The goal of the application is to provide the well-known services listed above but all of these have to work without an active internet connection and the application also needs to support older mobile devices.

The other goal of this application is that it can be used everywhere in the world where the schedule is accessible and downloadable in GTFS (General Transit Feed Specification) format [4].

2. Related work

There are many referenced public transportation and route planner systems. The first example is OneBusAway [5], which works on smart mobile devices, but it needs internet connection. This application shows static route maps and timetables to the users, and it also has a real-time tracker and trip planner functions. The application has a multi-tier architecture, so the trip planning happens on the server-side. It gives transit information about some USA regions.

Jariyasunant et al. [6] has also introduced a mobile transit trip planner, which works with real time data. Because of the real-time data, their application also needs internet connection to plan a trip.

Google Transit [2] is a very good trip planner, but it is online, so the smart mobile device user cannot use it without an internet connection. The other problem is that it knows only a few Hungarian cities, like Budapest, but it has no information about Debrecen.

The iBart application [7] is a transit application for iPhone. It works without internet connection. It includes station information, scheduled arrivals, a map, service advisories and a trip planner. You can plan trips with it if you are in the Bay Area. It works with GTFS data, in this way it is very easy to adopt to other cities. It has some other sister apps for transit systems in Boston, Chicago, London, etc [8].

On Google Play you can download transit schedule and trip planner applications which works with Hungarian GTFS data. The first example is Smart City Budapest Transport [9] which is free and works offline. It works only in Budapest based on the official GTFS data. Its trip planner is correct but slow. The other example is

BpMenetrend [10], which can't plan trips, but it stores the schedules of routes of Budapest. You can use it offline.

3. Our application

Our application lists the departure times, journey times and the stops of the lines. It can display the lines on the map. It shows the stops nearby with the line passing through. The user can choose one or more stops and can view the lines passing through the stops.

The application works with a data structure which previously built from a GTFS structure.

The main function of the application is the route planner. It plans at least one route from a starting-point to a destination. Before the route planning the user can set parameters which are taken into account by the route planner.

In the next sections we introduce the GTFS structure briefly, the data structure which is built from the GTFS structure, and is used by the application. Then we presents the conversion between the GTFS and our data structure. At last we give a short description about the route planner.

4. General Transit Feed Specification (GTFS)

GTFS is the de-facto standard for describing public transit data. It was designed by Google and TriMet. "It allows public transit agencies to publish their transit data and developers to write applications that consume that data in an interoperable way." [4]

We can think of it as a relational database. It was designed for server-side processing. Usually transit agencies convert their existing data to meet the requirements of the GTFS, and since the specification contains only a few constraints, the result are somewhat different for each city. GTFS feeds are relatively big, for example the size of the GTFS feed for Budapest is about 150MB. For these reasons, GTFS is not suitable for processing on mobile devices.

GTFS structure is open source, simple, unified and easy to understand. GTFS data can be easily acquired, you can download the zip file from many sources. Public GTFS data is accessible from the website of GTFS Data Exchange [11] and Google Transit Data Feed [12].

Our application provides transit schedules and route planner for Hungarian cities: Debrecen, Budapest and Nyíregyháza. GTFS data for Budapest can be downloaded from the website of Centre for Budapest Transport [13]. The GTSF data of Debrecen and Nyíregyháza is developed by a Debreceni Regionális Közlekedési Egyesület [14].

5. Data Structure

From GTFS data we build a data structure which can be easily stored in the mobile device. The program logic reaches the data based on the logical data structure, meanwhile the physical representation presents the way the data is stored.

5.1. Logical data structure

Figure 1 shows the main parts of the logical data structure. It is constructed using plain old data (POD) types, which means we used C primitive types (including pointers), structures and arrays.

Routes are organized into categories, which serve the purpose of grouping similar routes together. An example for the category is the bus, while an example for the route is the bus number 11 in Debrecen. A route always has at least one line, but usually it has at least two: one for the inbound travel and one for the outbound travel. If a route has a special path (e.g. when the bus goes to the garage at the end of the day), this path will also be a new line. For example the bus number 11 has 4 lines: 1. from Doberdó Street to Borzán Gáspár Street, 2. back on this path, 3. from Kenézy Gyula Kórház to Borzán Gáspár Street, and 4. back on this path.

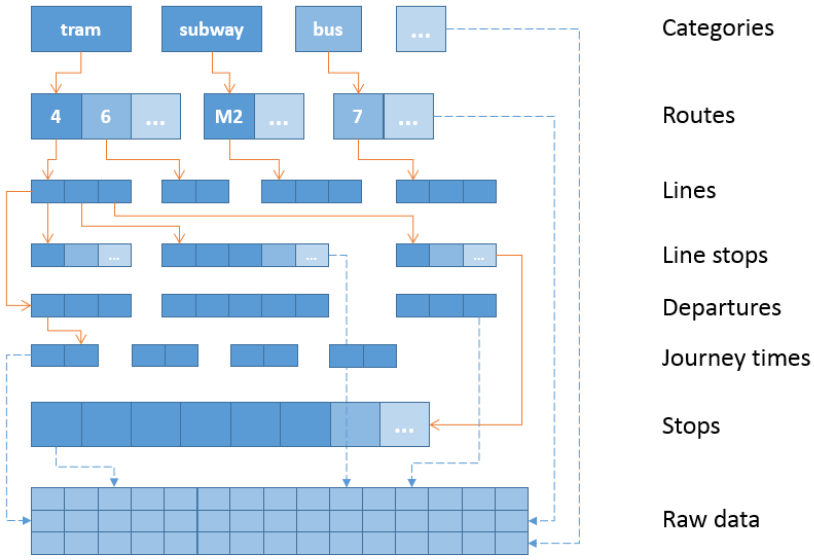

Figure 1: Logical Data Structure

For every line we store the stops that line goes through. Only departure times from the first stop are stored, the arrival and departure times for the subsequent stops are calculated from the journey times.

The database also contains data for public holidays, school periods, different types of days and shapes which is not shown on the figure.

5.2. Physical representation

Since our logical structure is built using POD types, it is straightforward to represent the data in memory using these simple constructs. While unique entities (like routes, stops, etc.) have identifiers, common entities (like line stops, journey times, etc.) do not have one, so they are accessed via pointers. Because there are a lot of duplicate data (for example, a name of a stop can appear multiple times, or there might be more than one lines with the same journey times), strings and almost all numeric data are stored in pool of raw data, and they are stored only once.

All of the data is placed at a continuous area in the memory, and all of the pointers also point to addresses inside this area. Because of this, we can save and load the whole database by writing this area to a file and loading the whole file again. We use two different methods for loading the database. The first method maps the whole file into the address space of the process using a predefined address (which is stored in the database). If it is not possible (for example, this region is already allocated for something else) we fall back to the second implementation which reads the file using regular I/O functions. In the latter case, pointers are needed to be adjusted to match the new addresses.

In order to maintain binary compatibility of the database, we use a wrapper class on 64 bit systems which translates the 32 bit absolute addresses to relative pointers.

6. Conversion from existing data

GTFS data is being processed and transformed on a desktop computer. The whole process is automated, and it consists of three phases.

Extracting data from the GTFS database: In the first phase GTFS data is downloaded and processed. We have developed a Java application for this task based on existing classes from OneBusAway [5]. This application loads the contents of the GTFS database into the memory and builds indexes to allow faster data access. After loading, the application tries to correct common errors and extrapolates missing data. The database is then validated and if no fatal errors occurred, the conversion process can go on. During the conversion the following data is being extracted: day types, route categories, routes, stops, shapes, lines, line stops, journey times, and departure times. The GTFS database might contain agency-specific fields that are taken into account during the extraction. As the result of this process an SQLite database is created whose structure matches the logical structure described above. This database does not contain any agency-specific tables or fields.

Adding additional data: In the second phase we enrich the database with some

extra information. Examples are public and school holidays, street and district names for bus stops, etc. The orientation of stops are also calculated using shapes and stops are being grouped using K-means clustering. This information does not exist in the GTFS databases but they are important for informing users. This phase is implemented using Python.

Building the physical representation: The last phase is performed by a third application written in C++, which is very similar to the database layer of the mobile application. This application loads the contents of the SQLite database and creates the necessary structures from the data. After all data is loaded, it allocates a continuous area in the memory and copies the structures, strings and arrays of primitive types into that area. Duplicate values are skipped, and pointers are adjusted to match the new addresses. This continuous area is being saved to a file as described in the previous section.

After the conversion process completes, the final database is uploaded to a server. Before the clients can download the database, it has to be checked for correctness and some metadata needs to be added (version, creation date, compatible application versions, etc.).

7. Route planner implementation

The route planner runs on the mobile device, so it was important to take the limited resources of the devices into consideration during the development of the algorithm.

7.1. Preprocessing phase

The algorithm starts with a preprocessing phase. During this phase, it calculates the arrival and departure times of vehicles for every stop for the next five hours (the size of the time window can be increased/decreased based on the size of the city). Thanks to our data structure this only takes a fraction of a second on a mobile device. Stops are also being indexed by their coordinates in order to make searching for nearby stops faster.

8. Planning phase

The goal of the planning phase is to find the earliest possible arrival time to the destination. This is achieved using an A algorithm [15]. There is a one-to-one relation between stops and states. The cost function simply returns the elapsed time from the start, and the heuristic function gives the shortest possible travel time to the destination, as if a direct connection would always exist (optimistic estimation).

The user can specify any two points to plan between, but to find a path, it is enough to take stops into account. First, the algorithm finds the closest N stops

to the start position, and calculates the amount of time needed to travel there by foot. These stop-time pairs will form the set of the start states. Similarly, the N closest stops to the destination are also found, and they will be the goal states.

The algorithm uses only one operator, which can return a number of new states. This operator searches for other stops within a given distance to the current stop. For every nearby stop it calculates the time required to walk there (arrival time). After that, for every line that goes though that stop it finds the first vehicle we can reach. Using all of the subsequent stops of that vehicle, the operator returns a number of new states which will be extended using this method, until the goal is found (or there are no more unvisited stops).

In order to reduce the number of states, we utilize an important optimization: a new state is only created if it has a chance to yield better results than any of the existing states. In order to achieve this, the algorithm uses a second heuristic function. This function calculates the amount of time needed to walk from a given stop to the destination. The algorithm keeps track of the current minimum value of this function for the whole process. During the extension process, if we encounter a stop for which the first heuristic function is greater than the current minimum of the second heuristic function (so the most optimistic estimation for the remaining travel time from that stop is worse than the most pessimistic estimation for the current best stop) that stop is simply skipped.

If the algorithm does not end in a goal state (because the destination cannot be reached by public transport), the state whose stop is the closest to the destination is selected. The last step of this phase is to walk from the goal to the destination location (if those two are not the same).

8.1. Fine tuning phase

The previous phase results in a path with the earliest arrival time, but it is not necessarily the best for the user. In this phase the algorithm tries to simplify this path by doing the following adjustments: optimizing the location of transfers to reduce the amount of walking; replacing short travels by walking if the waiting time for the next transfer allows it, and vice-versa; substituting parallel lines with each other to increase the number of possibilities.

The results of this phase are briefly presented to the user, who can select the best path for their purposes.

9. Offline maps

The application contains an offline map based on the Google Maps Android API v2 [16], which requires a persistent internet connection to work. In order to provide full offline functionality, we implemented a custom tile provider that can load and render tiles from a local database. This database contains the map data in vector format, so it is relatively small. Tiles are rendered in a background thread when needed, and the final tiles are cached on the external storage of the device in PNG

format. The map database is downloaded and updated the same way as the transit database, and the cache is automatically flushed after an update.

The tile loading and rendering code is based on MapsForge [17] with minor modifications to make it able to render tiles of arbitrary sizes. This was needed because the Google Maps API expects different tile sizes for different screen densities. The map data comes from OpenStreetMap [18].

10. Automatic updates

We decided to implement automatic updates using push notifications, which means that when an updates becomes available, the server send a broadcast message to all affected clients. This implementation has several benefits. First, it allows the application to install updates immediately as they become available, or when the device connects to the internet. Second, the application does not have to run continuously on the device just to check for updates. Last but not least, the server is also able to manage its own resource usage, since updates only happen if the server requests them.

11. Future work

In the future, we could use push notifications to provide more frequent (even hourly) updates. To minimize bandwidth usage and encourage users to actively use this feature, we will need to implement incremental updates. Frequent, but small updates will help us to provide accurate timetables even after unexpected transit changes, while retaining all of the functionality for offline users. (The frequency of updates are currently limited by the transit agencies).

12. Conclusion

The application can be downloaded from Google Play for Budapest, Debrecen, Nyíregyháza and Veszprém. The application is realized for four cities, the names are Budapesti Menetrend [19], Debreceni Menetrend [20], Nyíregyházi Menetrend [21] and Veszprémi Menetrend [22]. It can be ported to every city where a GTFS feed is available. You can find some useful information about the application on its website [23]. Plenty of user feedback can be found on Google Play. The users like to use the applications, as they make their life easier.

References

[1] Valós idejű vonatinformációk az Interneten. 2009 Retreived from: `http://www.mav-szk.hu/sajto/archivum.php?xquery=gps&mid=14b41c4c4a5adb`.

[2] Google Maps. 2014. `http://maps.google.com`

[3] Route Planner of BKV. 2014 http://www.bkv.hu

[4] GTFS Specification. 2014 Retrieved from https://developers.google.com/transit/gtfs/reference?csw=1

[5] Brian Ferris, Kari Watkins, Alan Borning: OneBusAway: A Transit Traveler Information System. MobiCASE 2009, LINCST 35, pp. 92-106. 2010.

[6] Jerald Jariyasunant, Daniel B. Work, Branko Kerkez, Raja Sengupta, Steven Glaser and Alexandre Bayen: Mobile Transit Trip Planning with Real–Time Data. UC Berkeley: University of California Transportation Center. 2011. Retrieved from: https://escholarship.org/uc/item/51t364vz

[7] iBart. https://itunes.apple.com/us/app/embark-ibart-san-francisco/id288656960?mt=8

[8] Wade Roush: Welcome to Google Transit. Community Transportation Assosiation. 2012 Retrieved from: http://web1.ctaa.org/webmodules/webarticles/articlefiles/Spring_12_DigitalCT_Google_Transit.pdf

[9] Smart City Budapest Transport https://play.google.com/store/apps/details?id=hu.ponte.mobile.smartcity

[10] BpMenetrend. https://play.google.com/store/apps/details?id=hu.bpmenetrend.activity

[11] GTFS Data Exchange. 2014 http://www.gtfs-data-exchange.com/agencies

[12] Google Transit Data Feed. 2014 https://code.google.com/p/googletransitdatafeed/wiki/PublicFeeds

[13] Official GTFS data for Budapest. 2014 http://www.bkk.hu/apps/gtfs/

[14] Website of Debreceni Regionális Közlekedési Egyesület. 2014 http://www.derke.hu/

[15] Stefan Edelkamp, Stefan Schrödl: Heuristic Search. Elsevier, 2012.

[16] Google Maps Android API v2. https://developers.google.com/maps/documentation/android/

[17] MapsForge. https://code.google.com/p/mapsforge/

[18] OpenStreetMap. http://www.openstreetmap.org/

[19] Budapesti Menetrend https://play.google.com/store/apps/details?id=hu.donmade.menetrend.budapest

[20] Debreceni Menetrend https://play.google.com/store/apps/details?id=hu.donmade.menetrend.debrecen

[21] Nyíregyházi Menetrend https://play.google.com/store/apps/details?id=hu.donmade.menetrend.nyiregyhaza

[22] Veszprémi Menetrend https://play.google.com/store/apps/details?id=hu.donmade.menetrend.veszprem

[23] Website of our applications. 2014 http://menetrend.donmade.hu/