String Charter 2: Visual Transport Schedules

Inge Gsellmann, Michael Hebesberger, and Danijela Lazarevic

706.057 Information Visualisation 3VU SS 2023
Graz University of Technology

03 Jul 2023

Abstract

String Charter 2 is a web application to generate interactive visual transport schedules in the form of string charts. This report provides an overview of the project, including an exploration of string charts as a visualisation method, the selection of the development framework, data formats and datasets, the project's implementation, and potential future work.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Being able to visualise public transport schedules is useful not only for passengers, but also for the agencies running the service. One could argue that it is even more useful for agencies, as they have more incentive to look at all available connections than single passengers, who might be satisfied with looking at two or three connections at a time.

Agencies can analyse their service for gaps in the schedule, or for track sections which consistently cause delays. Instead of reading through endless tables, trying to find patterns by hand, then find outliers or breaks in the pattern, having access to a data visualisation lets analysts actually see the patterns and gaps.

The graphic train schedule based on a string chart (or stringline chart) is a classic visualisation method for railway schedules and traffic. It is featured on the front cover of Edward Tufte's book The Visual Display of Quantitative Information. The string chart is often attributed to Étienne-Jules Marey, but was invented by Charles Ibry, an engineer with the French Northern Railroad company, to show train movements on a specific railway route throughout the day. The diagram displays the geographical route on the vertical axis and the hours of the day on the horizontal axis, allowing for a clear visualisation of train destinations and speed. An image of the route from Paris to Bologna from 1852 can be seen in Figure 1.1. The method was crucial for managing traffic on early railway routes, which often operated on a single track in both directions [Rendgen 2019].

String charts excel at visually presenting both time and trips, making them highly suitable for depicting public transport schedules. By mapping different trips on a single chart, commuters can easily discern the relationship between various transportation options. The two axes represent time and stops, resulting in an intuitive chart to read. The lines connecting points on the chart symbolise the movement of vehicles along specific routes, enabling users to quickly identify the frequency and duration of trips. Stringline charts can include information such as different modes of transportation, timings for peak and off-peak hours, and even gaps or delays.

Stringline charts are user-friendly and accessible to a wide range of users. They eliminate the need for deciphering complicated timetables or reading extensive text-based schedules. Moreover, stringline charts can be adapted for digital use. This ensures that users can access interactive information about public transport schedules, enhancing their overall travel experience [Lee and Multer 2009].

This project aims to create a simple, web-based application to convert a General Transit Feed Specification (GTFS) file into a string chart, which can then be exported as an SVG file for further use. Chapter 3 describes the GTFS file format and the Tauri and Electron frameworks for building executable packages from web applications.. After that, Chapter 2 looks at some similar projects and tools. Chapter 4 describes the requirements and implementation of the project. Chapter 5 presents some iodeas for potential future work. Finally, Chapter 6summarises the lessons learned and conclusions from the project.

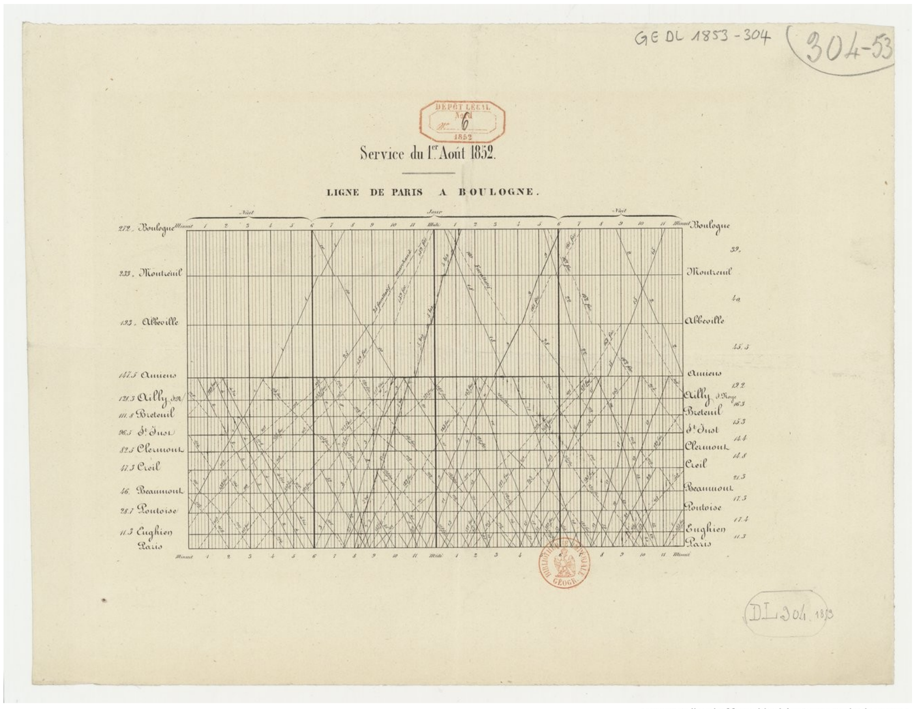


Figure 1.1: Image of a string chart from 1852 showing the connections from Paris to Bologna. [gallica.bnf.fr / Courtesy Bibliothèque nationale de France]

Chapter 2

Related Work

There are two notable projects that share similarities with String Charter 2: Marey's Trains from Observable [Bostock 2021] and the NYC Subway Stringlines from Vibien [Vibien 2019]. Both visualise transportation schedules in an interactive manner using string charts.

Marey's Trains visualises the transit lines from San Francisco to Gillroy, as can be seen in Figure 2.1. They use "baby bullets" to represent train stops. When hovering over a baby bullet, the graph displays additional information such as the line number, station name, and arrival time. Users can also choose to display the transit data for weekdays, Saturdays, or Sundays. In addition, the project provides users with the option to select the direction of transit, whether it's northbound, southbound, or both.

The NYC Subway Stringlines created by Vibien, visualises schedules on the New York City subway system, as can be seen in Figure 2.2. The most notable feature of this project is that it utilises real-time data and the chart is updated accordingly. The user can choose between northbound and southbound trips, as well as what subway line is being displayed. Additional features include displaying the dwell time, run time, headways, travel time, and trip IDs. The NYC Subway Stringlines incorporates sliders for setting the end date, end time, and the number of hours to be displayed for a route.

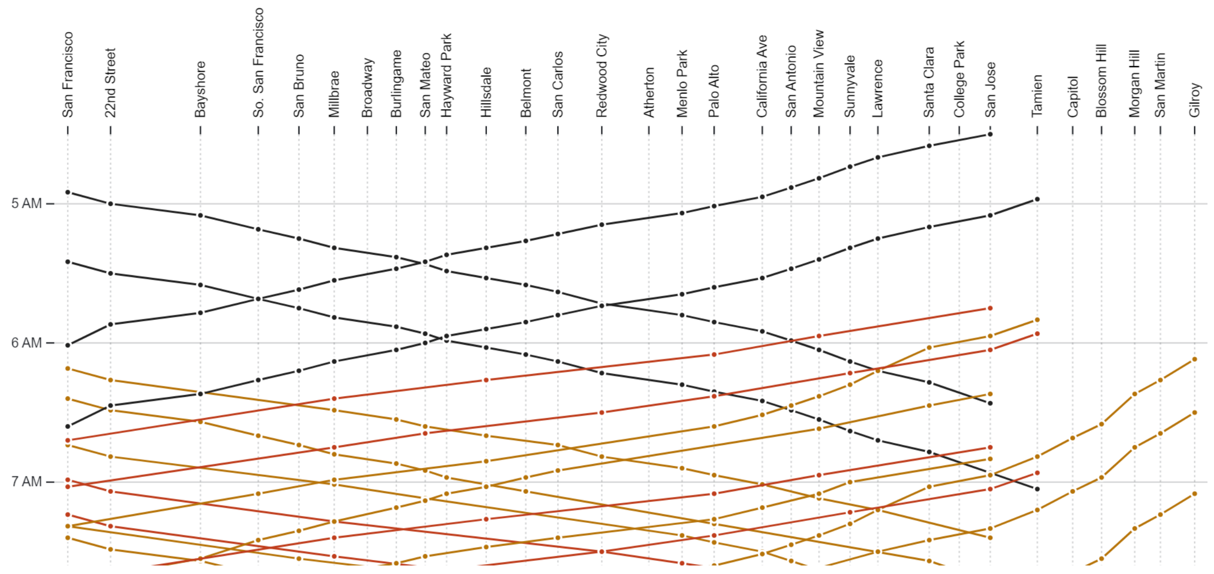


Figure 2.1: Marey's Trains string chart from Bostock [2021].

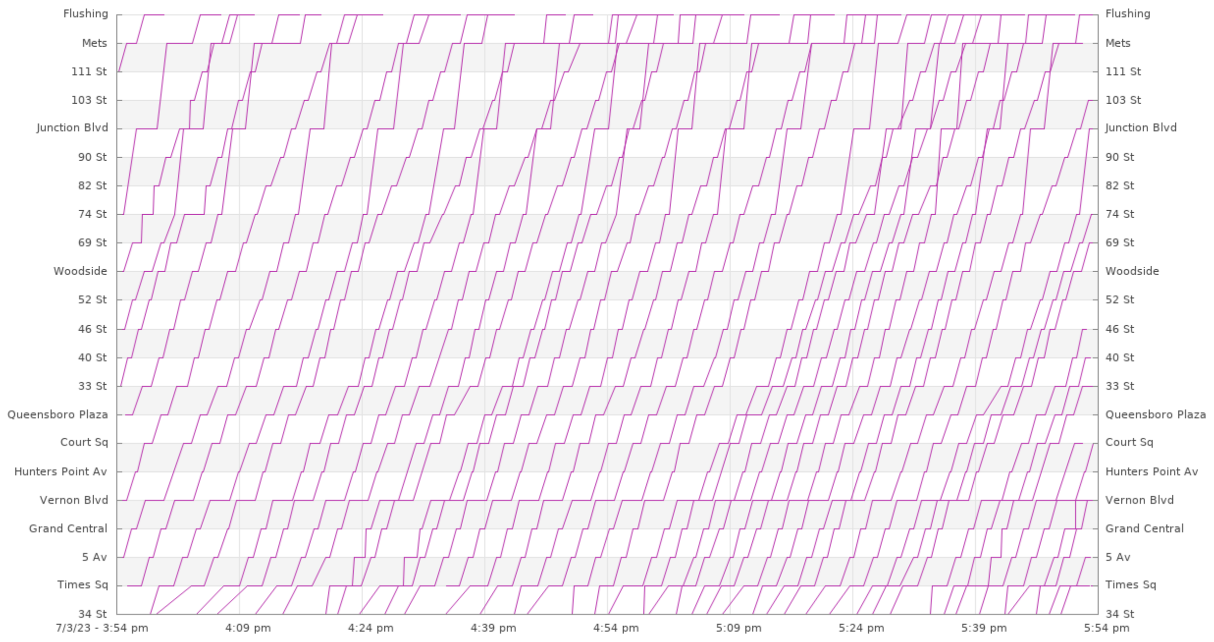


Figure 2.2: NYC Subway Stringlines from Vibien [2019],.

Chapter 3

Technology

This chapter looks at some of the technology behind the project, including the data format GTFS and the Tauri and Electron frameworks for building executable packages from web applications.

3.1 General Transit Feed Specification (GTFS)

The General Transit Feed Specification (GTFS) format is used to publish information on transit data, so it can easily be read by different tools[MobilityData 2019]. It was originally developed by Google, so as to have a unified data format usable by many different transit agencies, who at the time all used different formats. The format is published under an Apache 2.0 license.

There are two versions of GTFS: GTFS Schedule and GTFS Realtime. The former is for static schedules and is used in this project. The latter is meant for real-time updates of existing schedules, and covers delays, cancellations, and similar events.

GTFS Schedule is a data format where each file is a zip archive containing multiple `txt` files. The `txt` files follow a specified naming convention and contain comma delimited CSV data. Table 3.1 shows which files exist within the GTFS Schedule format and whether they are required or not. The table also shows the required fields for each file, although there are also many more optional fields and some fields which are conditionally required if other files or fields are present.

Of the five required files, the first and simplest is `agency.txt`, which contains information on all transit agencies running a service in this dataset. The `stops.txt` file defines the geographic information of stops by longitude and latitude and can also define entrances. `routes.txt` identifies transit routes containing groups of trips, which are themselves defined in `trips.txt` as a sequence of two or more stops in a specific time period. In order to put it all together into a proper schedule the last required file is `stop\_times.txt` which holds the data on when a certain trip will arrive or leave a each stop [MobilityData 2019]. Section 4.4 describes in detail how the different files in the GTFS Schedule format were used in this project.

3.2 Tauri

Tauri is a framework for building native desktop applications using web technologies such as HTML, CSS, and JavaScript or TypeScript [Tauri 2023]. It aims to provide a highly flexible and efficient development environment while leveraging the power and familiarity of web technologies.

One of the key advantages of Tauri is that a web application can be developed, but as well as running on a web server, it can also be built as executable packages for desktop operating systems such as Windows, macOS, and Linux. Tauri offers good performance by utilising a lightweight and minimal runtime. It minimises the overhead associated with running web technologies within a native environment, resulting in fast and responsive desktop applications.

File Name	Required	Required Fields
agency	✓	agency_name, agency_url, agency_timezone
stops	✓	**stop_id**, stop_name, stop_lat, stop_lon
routes	✓	**route_id**, route_short_name, route_long_name, route_type
trips	✓	**trip_id**, *route_id*, *service_id*
stop_times	✓	*stop_id*, *trip_id*, arrival_time, departure_time, stop_sequence
calendar	✶	**service_id**, monday, tuesday, wednesday, thursday, friday, saturday, sunday, start_date, end_date
calendar_dates	✶	*service_id*, date, exception_type
fare_attributes	✗	**fare_id**, price, currency_type, payment_method, transfers
fare_rules	✗	*fare_id*
fare_media	✗	**fare_media_id**, fare_media_type
fare_products	✗	**fare_product_id**, amount, currency
fare_leg_rules	✗	*fare_product_id*
fare_transfer_rules	✗	fare_transfer_type
areas	✗	**areas_id**
stop_areas	✗	*area_id*, *stop_id*
shapes	✗	**shape_id**, shape_pt_lat, shape_pt_lon, shape_pt_sequence
frequencies	✗	*trip_id*, start_time, end_time, headway_secs
transfers	✗	transfer_type
pathways	✗	**pathway_id**, *from_stop_id*, *to_stop_id*, pathway_mode, is_bidirectional
levels	✶	**level_id**, level_index
translations	✗	table_name, field_name, language, translation
feed_info	✗	feed_publisher_name, feed_publisher_url, feed_lang
attributions	✗	organization_name

Table 3.1: Overview of the files in GTFS Schedule format. A star means that there might be conditions in which a file is required. **Bold** fields are primary keys and *italic* fields are foreign keys.

Tauri offers the usage of popular frontend frameworks such as React, Vue.js, and Angular, allowing developers to work with well-known and well-supported tools and libraries. As of now, Tauri only supports Rust as their backend language and will not allow to build an executable without at least implementing a single function, even if it is never called.

3.3 Electron

Similar to Tauri, Electron.js is a framework which enables developers to build cross-platform desktop applications using web technologies such as HTML, CSS, and JavaScript [OpenJS 2023a]. Electron is built on Chromium and Node.js, providing developers with a robust and reliable runtime environment. Bundling Chromium and Node.js comes at the cost of performance and large file sizes when compiling the application to an executable.

By utilising Chromium, developers can create applications with rich web capabilities, taking advantage of modern web technologies and APIs. Additionally, Electron.js allows direct access to the underlying

Feature	**Electron**	**Tauri**
Architecture	Chromium	Native web renderer
File Size and Performance	Large file sizes and performance penalty due to overhead of full browser.	Small file sizes and better performance due to native webview components.
Frontend Languages	HTML/CSS/JavaScript, React, Angular, Vue	HTML/CSS/JavaScript, SvelteKit, Qwik, Next.js, Vite
Backend Languages	No specific backend language requirement	Rust
Open Source	Yes	Yes
Security	Security measures inherited from Chromium.	Strong focus on security.

Table 3.2: Comparison of Electron and Tauri.

operating system through Node.js, enabling developers to interact with system resources and use native functionalities, at the cost of their applications no longer running on the web.

Table 3.2 compares Tauri and Electron. For this project, due to its better performance and file sizes, Tauri was chosen to produce desktop executable packages.

Chapter 4

Implementation

This chapter describes the requirements and implementation of the String Charter 2 project, including what tools and datasets were used and why certain decisions were made.

4.1 Requirements

The requirements for this project were very clear from the beginning: an easy and fast way to load a static GTFS Schedule dataset and select a route from it in order to display it in a string chart. The user should be able to interact with the displayed routes and gather additional information upon hovering over the drawn baby bullets. In addition to drawing the chart in the application, the possibility to export and download the charts as an SVG file was very important as well. In order to make the chart as useful as possible, another important factor was to create non-overlapping labels. Since stations can sometimes have very long names, readability had to be a feature of our application.

Another important requirement was to make the application as accessible as possible. Whilst the main goal was to build a web application, desktop platforms should be supported as well, whilst also being as responsive as possible. This ensures that the app can be used by a wide range of users, without them needing to have any prerequisites other than their computers.

4.2 Tools and Libraries

The development of the project involved the use of several tools and libraries. Visual Studio Code was used as the development environment [Microsoft 2023b]. The creation of the desktop application was done in Rust [Rust 2023] with Tauri [Tauri 2023]. The backend and frontend were both developed using TypeScript [Microsoft 2023a]. Node.js was used for the development of the server-side application [OpenJS 2023b]. The management of dependencies and packages was handled using npm, the package manager for Node.js, simplifying the installation and management of packages [GitHub 2023]. Additionally, various libraries were integrated into the project, including JSZip for working with GTFS archives [Knightley 2023] and Papa Parse for parsing and handling CSV data [Holt 2023]. Table 4.1 provides an overview of the tools and libraries used, including their versions and a brief description of their functionality.

4.3 Datasets

Two main datasets were used in this project: the GTFS archives for ÖBB and for FlixBus, two prominent transportation companies.

ÖBB (Österreichische Bundesbahnen) is the national railway company of Austria [OEBB 2023]. The ÖBB GTFS dataset consists of the CSV files listed in Listing 4.1. It contains data on 323 routes, collected

Tool/Library	Version	Description
Visual Studio Code	1.78.2	An integrated development environment with a wide range of features and support for various programming languages.
Cargo	0.2.3	A tool for managing Rust projects, providing functionalities such as dependency management and building.
CodeLLDB	1.9.2	A debugger extension for Visual Studio Code.
rust-analyzer	0.3.1541	A language server for Rust, providing code analysis and other helpful features for Rust development.
Tauri	0.2.6	A framework for building desktop applications using web technologies. See Section 3.2 for more information.
Node.js	18.16.0	A JavaScript runtime environment which allows running JavaScript code outside of a web browser, enabling server-side and command-line applications.
npm	9.6.7	The package manager for Node.js, facilitating the installation and management of JavaScript packages and dependencies.
Rust	1.62	A programming language known for its memory safety and performance, suitable for building fast and reliable applications.
TypeScript	4.8.2	A superset of JavaScript which adds static typing and other features.
JSZip	3.10.1	A library for working with ZIP archives in JavaScript, providing functionalities for compression, extraction, and manipulation.
Papa Parse	5.4.1	A CSV parsing library in JavaScript, enabling easy parsing and handling of CSV data in various formats.

Table 4.1: Overview of the tools and libraries used.

```
agency
calendar
calendar_dates
fares
frequencies
pathways
routes
shapes
stop_times
stops
transfers
trips
```

Listing 4.1: The files included in the ÖBB GTFS dataset.

```
agency
calendar
calendar_dates
feed_info
routes
stop_times
stops
transfers
translations
trips
```

Listing 4.2: The files included in the FlixBus GTFS dataset.

in 2023. It should be noted that this dataset contain some errors, specifically instances where completely different trips were identified within a single route.

FlixBus is a well-known long-distance bus service provider operating across various countries in Europe [FlixBus 2023]. The FlixBus GTFS dataset consists of the CSV files listed in Listing 4.2. It contains data on 528 routes, collected in 2021. Unlike the ÖBB dataset, the FlixBus dataset does not exhibit errors such as completely different trips being identified within a single route.

4.4 Parsing

The parsing step involves extracting the relevant information from the GTFS files and creating structured objects to accurately represent the transit data. These structured objects are essential for displaying string charts, since they encompass transit lines, all trips within each transit line, stop names, and arrival times. Figure 4.1 illustrates the GTFS data model, as required for this project.

To begin, the transit lines can be found in the route file. Each transit line possesses a unique identifier, which serves as a key connecting the trips file to their respective routes or transit lines. Similarly, every trip is assigned a distinctive identifier, which is then employed in the stop times file to establish the linkage between arrival times and specific trips. Furthermore, each arrival time is uniquely identified and can be utilised to determine the corresponding stop name within the stops file.

In order to effectively manage the extensive linking required and efficiently handle the potentially large GTFS files, the development of a robust parser becomes imperative. Looking at Figure 4.2, one can observe the structured representation of the parsed data. This final structure is organised as follows.

The parsed data is encapsulated within an array, with each element representing a route. Each `route` object within the array contains three attributes:

- `id`: The unique identifier for the route.
- `name`: The name of the route (for example Graz -> Vienna).
- `trips`: All the trips associated with the route.

Within the `trips` array, each trip object possesses five key properties:

- `id`: The unique identifier for the trip.
- `name`: The head sign of the trip, which is the first station in a trip.
- `routeId`: The route ID which the trip belongs to.

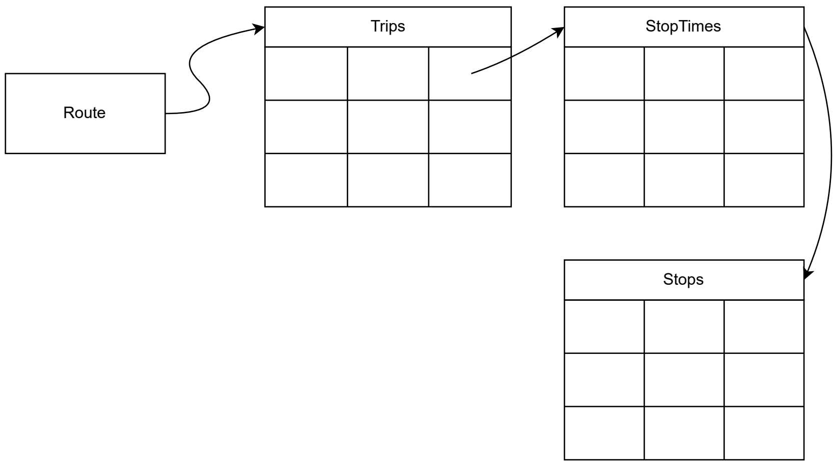


Figure 4.1: The data model of GTFS, showing the 4 files relevant for string charts. [Image created by Danijela Lazarevic using `draw.io`.]

- `stations`: The names of all the stations along the trip.
- `stops`: All the stops within the trip.

For each `stop` object within the `stops` array, the following four attributes are defined:

- `id`: The unique identifier for the stop.
- `tripId`: The trip ID the stop belongs to.
- `station`: The of the station.
- `arrivalTime`: Arrival time at the particular stop.

By adopting this structured approach, the parser ensures a coherent representation of the GTFS data, enabling efficient storage, retrieval, and analysis of the transit information.

The algorithm to fill the created structure unfolds in the following steps:

1. *File Loading*: The JSZIP library is used to load the GTFS archive asynchronously. It extracts the required files: `trips`, `routes`, `stop_times`, and `stops`.
2. *Parsing Route Data*: The `routes` file is parsed using the Papa Parse library. This process extracts the route ID and route name, both of which are stored in the route object. Furthermore, an empty array is initialised to hold the associated trips, which will be populated subsequently.
3. *Parsing Trip and Stop Data*: The `trips`, `stop_times`, and `stops` files are parsed. Three record data structures are initialised to store the trips using the route ID as key, the station names using the stop ID as key, and the stops using the trip ID as key. By using record data structures, the code achieves efficient data organisation and retrieval. They allow for direct access to specific data elements based on their associated identifiers, reducing the need for iterating through large arrays.
4. *Linking Data*: The routes array is iterated over and each route's `trips` array is populated with the associated trips. The relevant stop times are also assigned to each trip matching the trip ID.

This approach allows the parser to efficiently parse and link the data. The resulting data structure is illustrated in Figure 4.2.

While doing the parsing in Rust would have been more efficient, TypeScript was used to allow the application to work as a web application with cross-platform support, rather than solely providing desktop packages. Using Rust with built-in Tauri methods means that the application would no longer function as a standard web app.

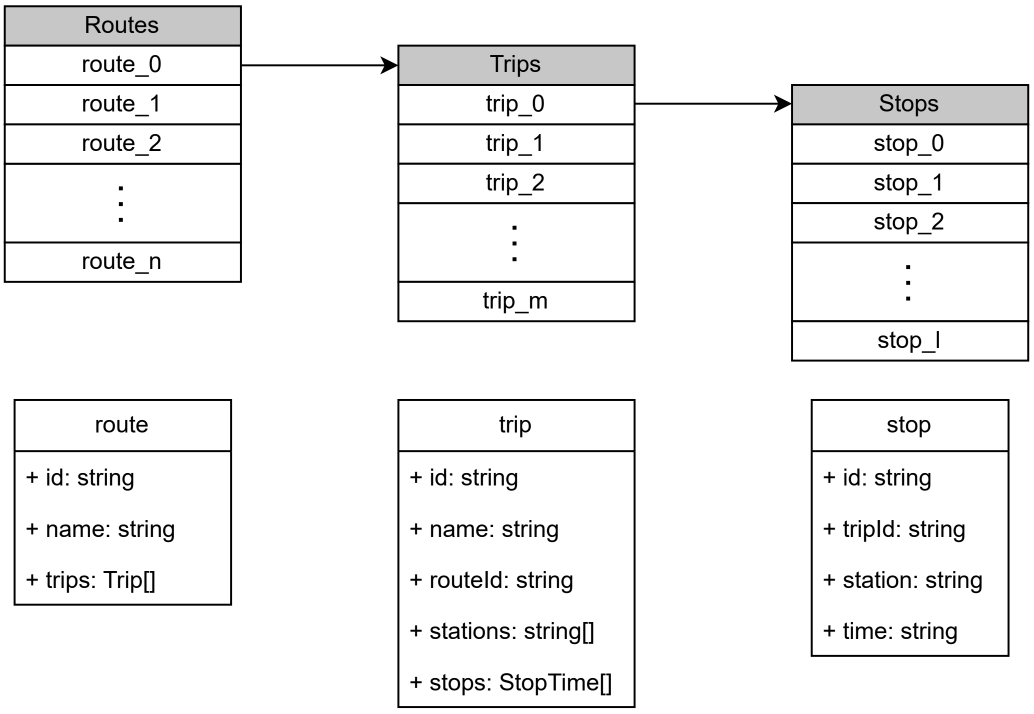


Figure 4.2: The structure of the parsed GTFS data. [Image created by Danijela Lazarevic using `draw.io`.]

4.5 User Interface

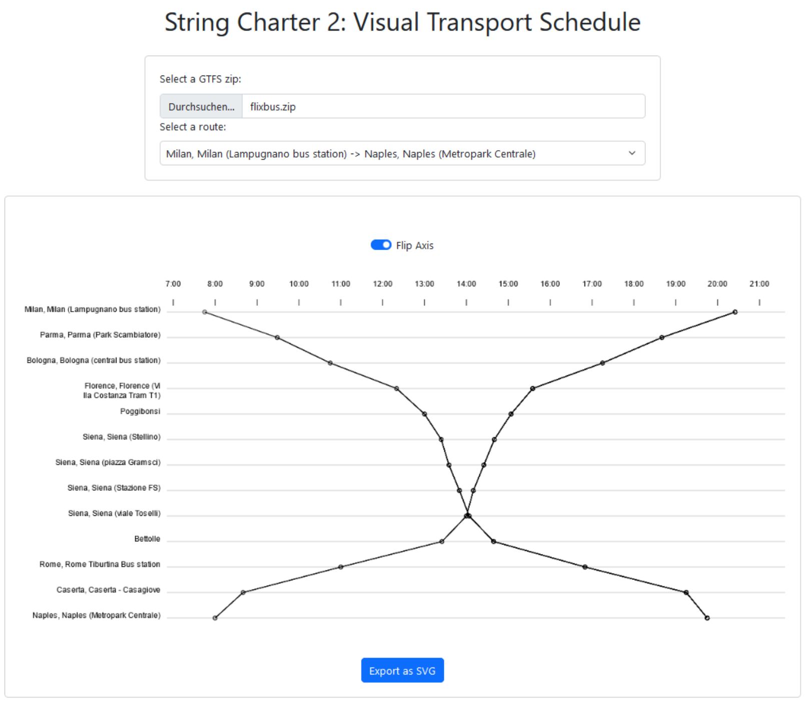
The user interface was kept as simple as possible. It is divided into a header with the name of the application, and two panels: one for uploading the GTFS zip archive and selecting routes, and one for the chart and its interactive components.

The user can upload a GTFS zip archive using the File Input element. After the upload a select element, which is located beneath the file-input, is populated automatically with all available routes. As soon as the user selects a route, the second card containing the chart and its interactive components becomes visible. The user now has the possibility to hover over the charts baby-bullets and see additional stop information at the very top of the card for the chart. Beneath the placeholder for this additional information exists a switch element to flip the axes of the chart. Finally, located beneath the chart is the button to export the drawn graph as an SVG file. In Figure 4.3 all the components can be seen. The styling was done using Bootstrap v5.2 and some custom CSS code.

4.6 Drawing the Chart

The chart is drawn on an HTML Canvas Element to enable interactive functions like mouseovers. Horizontal grey lines are drawn for each data value on that axis to aid the viewer in seeing where a particular data point lies.

There are different margins needed for the flipped axes because times involve significantly shorter strings than station names. If time is on the vertical axis, the station names on top are rotated by 45 degrees so there is no overlap of the names. Additionally, this leads the user to instantly see where in the chart the vertical column for a station is and makes it easier to identify which station a stop belongs to. If time is on the horizontal axis, this tilting of the label is not necessary. Instead, short ticks are added underneath the time labels to show the viewer where exactly the data points for this time are placed. Some station names are very long so when placed on the vertical axis any label longer than 40 characters is split into two lines.


String Charter 2: Visual Transport Schedule

Select a GTFS zip:
Durchsuchen... flixbus.zip
Select a route:
Milan, Milan (Lampugnano bus station) -> Naples, Naples (Metropark Centrale)

Flip Axis
7:00
8:00
9:00
10:00
11:00
12:00
13:00
14:00
15:00
16:00
17:00
18:00
19:00
20:00
21:00
Milan, Milan (Lampugnano bus station)
Parma, Parma (Park Scambiatore)
Bologna, Bologna (central bus station)
Florence, Florence (Vi
lla Costanza Tram T1)
Poggibonsi
Siena, Siena (Stellino)
Siena, Siena (piazza Gramsci)
Siena, Siena (Stazione FS)
Siena, Siena (viale Toselli)
Bettolle
Rome, Rome Tiburtina Bus station
Caserta, Caserta - Casagiove
Naples, Naples (Metropark Centrale)
Export as SVG

Figure 4.3: User interface of String Charter 2.

Trips gathered from the parsed and already filtered by route dataset are drawn with lines, with every stop being marked by a small circle. When hovering the mouse over those circles, additional information is displayed at the top of the graph, specifically the name of the route, the name of the stop, and the exact time of the stop. This is especially useful when a stop is scheduled not on the full hour but sometime in between, like most stops are. Otherwise, it can be quite difficult to reliably estimate a time from the chart.

At first, this detailed information was set to be shown in a small label box right next to the stop circle. However, the combination of Canvas elements and not sanity-checked data caused problems here, as the dataset contained multiples of the same (or slightly varying) trips, which were drawn with overlapping elements. This overlap caused too many function calls when moving the mouse over circles and especially when leaving the circles again, which caused the page to lag or even crash. Putting the detailed information outside of the chart eliminated the necessity for redrawing the graph on leaving the circle.

4.7 SVG Export

SVG generation is done separately to how the chart is drawn on the canvas, but follows a similar structure. The horizontal guiding lines, label placement and drawing of data entries is the same, taking into account the current status of the toggle to flip the axes, so the user receives an SVG in the same orientation as they are looking at on the screen.

Some alterations were made to compensate for different kerning and handling of margins. When generating the SVG with exactly the same values as the canvas elements, the result looks different. In particular offsets for labels changed the look of the chart, possibly because of slight variations in text alignment as well.

Station names are split into two lines if longer than 33 characters instead of 40 like on canvas. This was done because of kerning differences and to preserve the size of the chart itself, since the alternative would have been to widen the space available for labels, which would in turn reduce the space available for the chart.

All SVG elements are added as either text, line, or circle elements, which produces a much neater SVG than many online converters, which often only convert everything into strokes – even text. The generated SVG passes the W3 Markup Validation Service without any errors [W3C 1994].

Chapter 5

Future Work

There are numerous opportunites for further features right in the core functionality of the software as it works now, as well as a few ideas which would require new features to be added to the current code base. The points below are loosely ordered based on the amount of work they would involve.

When writing the SVG elements into the file, they could be grouped into different <<g>> elements, to make them easier to read for humans. For instance, there could be a group for each axis, or for each trip line, etc.

The list of routes in the dropdown menu could be ordered alphabetically so users can easily find a specific route they are looking for. Alternatively, the route selection could be split up into two input fields altogether, with the user picking their start point and destination separately.

Detailed station information, which is currently shown in a text field above the chart could be moved to small labels next to the currently selected stop. The reasons why this was not done in the current iteration of the project are outlined in Section 4.6.

Another function that could be added would be for the user to be able to click on a line and have the entire trip highlighted. This could also activate all detail windows for stops if that functionality was already added. This would make it easier to follow and get information on a single trip.

Information on the type of trip being displayed could be colour-coded in the graph. For instance bus connections, railjet trains, local transport, replacement buses, etc. could all have their own colour in the graph. This would vary a bit with each agency as they provide different types of connection.

Instead of just looking at complete routes, it would also be possible to let the user choose intermediate stations and receive a chart of just that segment, but from all possible routes. For instance, someone could pick Bruck an der Mur and Mürzzuschlag which are common stops in many routes but are not the endpoints of the routes themselves.

Rather than loading a GTFS file, there could also be an option to enter a URL where the file is found or even present a list of known agencies to choose from.

Many of the previously mentioned functionalities only really make sense on a properly sanitised dataset. Some datasets seem to contain multiple entries for the same trip or trips which do not cover the whole route, and other issues. See Section 4.3 for more details. Some of these issues are rather minor, but others fundamentally mess with the way string charts present data, or with the way the user interacts with it. To be able to properly display any given GTFS file without issues, the incoming data would need to pass a sanity check. This is not a trivial task for two reasons. Firstly, the possibility for errors and unintended use of the format are vast and would need to be catalogued, so that a sanity check can take all of them into account. Secondly, each case of unintended use would need to be resolved somehow, which would need to be decided on a case by case basis. The dataset might use the GTFS format in a way not intended by the makers of the format.

In addition to GTFS Static, the chart could also include GTFS Realtime information if available. This would require wholly different parsing as GTFS Realtime uses an entirely different structure. The feed would have to be updated live and any new information parsed immediately and included in the chart in some way which highlights the new information as a live change instead of a schedule change.

Chapter 6

Concluding Remarks

String Charter 2 successfully achieves the goal of parsing and visualising transit data using the GTFS format. The use of Tauri allows the web application to also be built as a native desktop application. The parsing algorithm efficiently extracts the necessary information from the GTFS files and organises it into structured objects, enabling a logical representation and easy retrieval of transit data. The user interface provides an interactive and user-friendly experience for exploring the string chart. The generated charts effectively display the transit lines, allowing the users to gain insights and analyse the data.

However, there is still room for improvements. Firstly, the current implementation focuses on the static representation of transit data. Integrating real-time information from GTFS Realtime feeds could enhance the application, by displaying live updates in the transit system. Secondly, while the application provides an interactive user interface, additional features such as search functions, filtering options, and highlighting of trips could improve the usability.

In conclusion, String Charter 2 demonstrates its capabilities in parsing and visualising transit data using the GTFS format. The application's cross-platform compatibility and efficient parsing algorithm contribute to its usability and effectiveness.

Bibliography

Bostock, Mike [2021]. *Marey's Trains*. 07 Oct 2021. https://observablehq.com/embed/@d3/mareys-trains (cited on pages 3–4).

FlixBus [2023]. *FlixBus GTFS - OpenMobilityData*. https://transitfeeds.com/p/flixbus/795 (cited on page 11).

GitHub [2023]. *npm*. https://npmjs.com/ (cited on page 9).

Holt, Matt [2023]. *Papa Parse - Powerful CSV Parser for JavaScript*. https://www.papaparse.com/ (cited on page 9).

Knightley, Stuart [2023]. *JSZip*. https://stuk.github.io/jszip/ (cited on page 9).

Lee, Mary T. and Jordan Multer [2009]. *Visualizing Railroad Operations: A Tool for Planning and Monitoring Railroad Traffic.* (Jan 2009). https://rosap.ntl.bts.gov/view/dot/8764 (cited on page 1).

Microsoft [2023a]. *TypeScript: JavaScript with Syntax For Types*. https://typescriptlang.org/ (cited on page 9).

Microsoft [2023b]. *Visual Studio Code*. https://code.visualstudio.com/ (cited on page 9).

MobilityData [2019]. *General Transit Feed Specification*. 18 Feb 2019. https://gtfs.org/ (cited on page 5).

OEBB [2023]. *Soll Fahrplan GTFS - Datensätze - ÖBB Open Data*. https://data.oebb.at/de/datensaetze~soll-fahrplan-gtfs~ (cited on page 9).

OpenJS [2023a]. *Electron*. OpenJS Foundation. https://electronjs.org/ (cited on page 6).

OpenJS [2023b]. *Node.js*. OpenJS Foundation. https://nodejs.org/ (cited on page 9).

Rendgen, Sandra [2019]. *Historical Infographics: From Paris with Love*. 15 Mar 2019. https://sandrarendgen.wordpress.com/2019/03/15/data-trails-from-paris-with-love/ (cited on page 1).

Rust [2023]. *Rust Programming Language*. https://rust-lang.org/ (cited on page 9).

Tauri [2023]. *Tauri*. https://tauri.app/ (cited on pages 5, 9).

Vibien, Philippe [2019]. *NYC Subway Stringlines*. 2019. https://pvibien.com/stringline.htm (cited on pages 3–4).

W3C [1994]. *Markup Validation Service*. 1994. https://validator.w3.org/ (cited on page 15).