# CS480 - Data-Visualization: Cartarail, Process Book

Authors:

- CHIPLUNKAR Shardul - 353675
- GUAN Yawen - 353858
- PINAZZA Alexandre - 282395

# Implementation

## Setup

Frontend: TypeScript, Vite, D3.js, AlaSQL, Homography
Data Processing: Python

## Data

### Preprocessing

#### Filtering

Calculating the shortest trips to **all** reachable stops across Switzerland requires substantial computational resources. To reduce this demand, we focus on two specific subsets:

- Swiss national rail network (3140 stations, 792 routes, about 1,700,000 schedule records);
- Lausanne metropolitan area (within 10km from Lausanne-flon, 573 stations, 125 routes, about 850,000 schedule records).

We filtered the original dataset, 2025 Switzerland public transport timetable, to include only the relevant stations, routes, and schedules within our selected area.

The code is written in python, available at notebooks/filter.ipynb.

#### Generating query tables

The original dataset is stored in the General Transit Feed Specification Format (GTFS) format, which is not optimized for the type of queries required by our shortest path algorithm. Specifically, the algorithm needs to repeatedly retrieve, given a start station and departure time, all reachable next stations along with their earliest arrival times.

To support efficient querying, we precomputed two query tables. The first is the *transportation table*, which stores all directly connected station pairs along with their

corresponding timetables. For each connection, the table includes: the start station, the next station, the route used, and all scheduled departure and arrival times (e.g., 08:00–08:40, 09:00–09:40, etc.) sorted by departure time.

The transportation table significantly reduces query time. For the rail network, it contains only about 18,000 records, each with all the necessary information, eliminating the need for the algorithm to repeatedly query the 1.7-million-record schedule table.

We also precomputed the walking times between all station pairs, considering only those under 30 minutes.

The scripts are written in python, available at scripts/, named gen-tranport-table.py and gen-walk-table.py.

### Loading

Since the query tables are small enough (about 20M in total), we chose to load the query tables (of the selected area) into the browser. We used a client-side in-memory SQL database AlaSQL for query.

## Shortest Trip Algorithm

Given a starting station and departure time, the shortest trip algorithm should return a set of reachable stations, each accompanied by its corresponding shortest trip. We implemented Dijkstra's algorithm, which finds the shortest paths in a weighted graph.

In our implementation:

- The reachable next stations from any given station include those accessible via public transport or on foot. We set a limit that the total walking time within any trip must not exceed 30 minutes.
- If two consecutive trips belong to different routes, one needs to transfer within the station. We apply a transfer time of 3 minutes between them.

To optimize the performance:

- We used a priority queue;
- Given two adjacent stations on a route with multiple scheduled trips, we assume that a trip that departs earlier will always arrive earlier. Therefore, given the sorted schedule, we can use binary search to find the trip with earliest arrival that departs after the given time.

The asymptotic complexity of the algorithm is $O(|E| + |V|\log|V|)$ (E: connections, V: stations), and the database queries take some time, the algorithm usually takes several seconds to run (depending on the start station, time, and the browser).

## Visualization

At a high level, the visualization uses the shortest trip data computed by the algorithm above to lay out a graph, where the vertices are stations, using a spring layout algorithm. The original and final positions of the vertices are used to warp the basemap image using a

triangle mesh mapping. Lastly, the UI is set up to allow the user to interact with the visualization to search for stations and see route information.

In the spring layout, there are three types of spring edges:

- Between the starting station and each destination, with preferred length proportional to the travel time. This makes visual distances reflect travel times.
- Between each destination and the previous station on the route to get there (if any), with preferred length proportional to the travel time on that last segment. Recursively, this means that there is an edge for every segment of every path, forming a tree rooted at the starting station. This keeps visual distances coherent between different destinations; if the path to one goes through the other, then along with the direct edges mentioned above, the two will tend to be collinear with the starting station.
- Between each station and its original, fixed location on the map, with preferred length zero. This makes the layout roughly match geography. Without it, the layout would be free to rotate or flip, keeping distances unchanged.

Of course, in an equilibrium state, not all edges will have exactly their preferred lengths. The balance between their conflicting forces depends on their strengths. The first two types of edges have the same strength as they represent the same kind of constraint. The third type of edge is weaker; we experimented with its strength until we found the right balance between geography and travel time. The simulation is implemented using D3.js and run for 10,000 ticks, which is experimentally more than enough for it to reach an equilibrium.

The initial vertex positions (corresponding exactly to the geography) are used to compute a triangle mesh over the basemap, using a Delaunay triangulation. Each point in the basemap can thus be identified by the triangle that contains it and its position inside that triangle in barycentric coordinates. Then, given new vertex positions, the basemap can be accordingly warped by mapping each point in the basemap to the same barycentric coordinates with respect to the new triangle. Four extra fixed triangulation points at the four corners of the basemap ensures that the warped basemap has the same dimensions. We also filter out vertices that are too far outside the bounds of the basemap. The triangulation and point-mapping is done using the Homography library.

In fact, we compute warped basemaps as above not just for the final vertex positions according to the spring layout, but also for 20 interpolated positions, letting us smoothly animate the warping.

Finally, with the help of D3.js, we set up the UI so that hovering over a vertex displays information about the path. The rest of the UI for selecting the scale, setting the starting station and time, and searching by station name is set up using standard JavaScript/TypeScript.

# Challenges and compromises

## Shortest Trip Algorithm

### Calculate walking time

We assume a constant walking speed of 4 km/h and estimate walking time between stations using only their longitude and latitude, without accounting for the actual walking paths or terrain. Similarly, we do not consider different platforms or tracks within a station—each station is treated as a single point for simplicity.

### Total walking time constraint

We limit the total walking time of a trip to 30 minutes. In the current implementation, each time a new connection is considered, we calculate the cumulative walking time based on the current shortest trip and check whether it remains within the 30-minute limit.

As a result, the algorithm tends to allow walking early in the trip until the walking budget is exhausted. However, this locally greedy approach may miss shorter overall trips that would be possible if some walking were deferred to the end instead.

Addressing this limitation is challenging as Dijkstra's algorithm is not well-suited for handling such global constraints. We consider that favoring walking earlier in the trip to be an acceptable approximation.

### Trips that arrive at the same time

It's common for multiple trips to arrive at the destination at the same time. For example, from EPFL to Allaman at 07:25, one will first reach Renens to catch the nearest train—R8, departing Renens VD at 07:39 and arriving in Allaman at 07:57.

To reach Renens in time, one could take the M1 metro from EPFL at either 07:28 or 07:33—both options are valid. In practice, we would prefer the 07:33 departure, as it allows for a later start.

However, for simplicity, the current algorithm does not distinguish between trips that arrive at the same time. It guarantees the earliest possible arrival at the destination given a start time, but not the latest possible departure.

## Visualization

Two challenges are inherent to such a visualization. The first is: starting from point A, what do we do when point B is further from point C in real life, but closer by travel time? Does it make sense to "fold" the map "over itself"? The second is: what do we do with points that

are not reachable at all, i.e. have “infinite” warped distance? We do not have good answers to these questions yet.

A third inherent challenge is that we need to pin the four corners to make the warped map the same shape as the original, but the boundary of the map is not actually a rectangle. It would make sense to pin more points closer to the real boundary, such that the map lies inside the polygon they define; in the limit, we would pin infinitely many points along the boundary. The optimal choice is somewhere in between but we do not know exactly where.

We notice that for some configurations, the warped image looks weird. Maybe this is also an inherent challenge, but maybe using a higher-resolution image, a different warping algorithm, better filtering for problematic vertices, etc. could lead to better results. We did not originally intend to animate the warping, but added that feature because it was hard to understand the warped map without it. Similarly, some numerical constants like the ratio between spring length and travel time (the “speed”) and the relative strengths of the two kinds of springs were set by manual experimentation, but we may not have explored the full space of possibilities.

Due to time constraints for implementation, when the user hovers over a station, we do not highlight the path visually but only show it in the table to the side.

Also due to time constraints for implementation, the user cannot zoom or pan the map. The warping computation is actually fast enough to allow on-the-fly warping of basemap images. We could take the same approach as any map navigation software and load different “tiles” of the map at different scales as the user interacts with the visualization.

# Peer Assessment

In our project, the design phase was a collaborative effort with contributions from all team members. The format of the preprocessed data was designed by the entire team, with Yawen implementing the preprocessing scripts. The path-finding algorithm was proposed by Alexandre and implemented by Yawen. The spring-layout algorithm used for the visualization was designed by Shardul and Alexandre, and Shardul implemented it in addition to the other UI elements. Each team member contributed equally to the project write-ups. Finally, Alexandre created the screencast.