Title slide

We present our project, *SDT Navigator*, a tool designed for efficient route planning in public transit networks.

What is this all about?

Our goal sounds simple: to compute the best possible journey from point A to point B. But what defines “best”? That’s a more complex question. It involves balancing multiple criteria like travel time, number of transfers, and convenience of the overall trip.

We came across a research paper on the subject. The authors described how to find Pareto-optimal journeys, meaning those that cannot be improved in one aspect (for example time) without worsening another (for example transfers).

We decided to apply their approach to a public transit application. One way to do this would be to find all the Pareto-optimal journeys and filter out the best ones using some cost function. However, we didn’t have enough time to implement the search with multiple criteria, so the only metric we considered was arrival time.

Demonstration

It’s show time!

Algorithms

RAPTOR

- The central object of the algorithm is a timetable – (S, R, T, F)
    - S – set of stops
    - R – set of routes. A route is an ordered sequence of stops a public transit route takes.
    - T – set of specific trips the routes take. A route may correspond to multiple trips, but a trip only corresponds to a single route. Each trip is a sequence of stops together with stop times.
    - F – set of foot-paths - transfers between nearby stations. Foot-paths are always available and take a fixed amount of time.
- The algorithm works in rounds. After the Kth round, the algorithm finds optimal journeys consisting of no more than K trips.
- On each round, the algorithm relaxes along all the routes. While traveling along the route, we maintain the earliest trip we could board until now, and relax arrival times accordingly.
- At the end of the round, arrival times are relaxed along foot-paths like in Ford-Bellman.

- Also, 2 heuristics are used: local pruning (we only consider stops that were relaxed during the last round, and routes passing through them) and target pruning (we don't consider some trips if we already know we can reach stop B faster)

Suffix automaton

- We also used the suffix automaton data structure to find stations by substring. For this, suffix automatons of all station names are joined together. This data structure is part of the course so we won't cover it in detail.

Data collection + Architecture

For testing and demonstration, we downloaded datasets of multiple cities in the GTFS format. They conform to the GTFS (General Transit Feed Specification), a common format for public transportation schedules. We used multiple public databases, such as gtfs.org and openmobilitydata.org.

Testing

Since both algorithms are separated from the rest of the logic, it was convenient to run them independently. We used multiple public transit routes in different cities as unit tests.
We also have stress tests of the suffix automaton, as well as property-based testing of the "triangle inequality" on random routes

Interface

We have a CLI interface in a REPL style. One of the datasets is preloaded on startup.
There are 2 commands:

- search – searches stations by substring
- route – finds the fastest route between two stations with a given departure time

We've also made a Telegram bot wrapper with the same interface.