BIMHD

semester project

Introduction

Our project aims to develop a REST API in Rust using the tiny_http REST API framework to process General Transit Feed Specification (GTFS) data of public transportation in Bratislava. The goal is to provide a comprehensive tool for accessing and utilizing public transportation data efficiently. Through this project, we aim to enhance our understanding of API development and data processing in Rust.

Requirements

We have set for ourselves the goals we would like to achieve in the development of this project:

- Displaying routes from stop A to stop B:
  - Based on specific time
  - Overall (without specific time)
  - Estimated Time of Arrival (ETA) (excluding walking)
- Listing available connections:
  - For a particular stop
  - Listing departures for connections at the given stop
- Listing nearest stops for a specific GPS location
- Development metrics - Return processing time within responses
- Result caching (in memory / filesystem)
- Ability to select specific types of vehicles

Design choices

Fuzzy search

While building our application, we wanted to implement fuzzy search for stops. Initially, we used Levenshtein distance-based libraries, which count the operations needed to transform one string into another. However, these libraries produced unsatisfactory results, often returning matches that didn't align with our expectations.

To improve this, we switched to a trigram-based fuzzy search library. This approach breaks strings into sets of three-character sequences, allowing for more flexible and accurate matching. The trigram-based method yielded faster search responses and provided better results, even with typos or alternate spellings. This led to a more responsive and user-friendly search feature, significantly enhancing the user experience.

Data indexing

To enhance the speed and efficiency of our search functionality, we implemented a variety of indexes, each serving a differnet purpose in optimizing trip routing and information retrieval.

Singular Trip Index

The first index was a HashMap designed to store information about every stop and the trips associated with it. While this index is not primarily used, it helped significantly in building a second index.

Trip Pair Index

The second index was focused on mapping between pairs of stops. This index used a tuple of start and target stop IDs as the key, with the value being a list of direct trips that included both stops in the specified order. This structure allowed us to efficiently identify all direct routes between two specific stops, reducing the complexity of searching for direct connections within large datasets.

Graph-Based Trip Index

The final index was a two-level map that helped with pathfinding. In this index, the key at the first level was the start stop, while the value was another map where the key was the destination stop. The values in this second-level map were the trips that connected these two stops. This structure was highly adaptable for implementing pathfinding, allowing us to explore various routing strategies with minimal computational overhead.

Path finding

1. **Dijkstra's Algorithm**: Dijkstra's algorithm finds the shortest path from a single source node to all other nodes in a graph. It iteratively selects the node with the smallest distance from the source and explores its unvisited neighbors, updating their shortest path distances until all nodes have been visited.
2. **A* Algorithm**: A* is an extension of Dijkstra's that includes a heuristic to estimate the cost from the current node to the target, optimizing the search process for a faster find. It selects the path that minimizes `f(n) = g(n) + h(n)`, where `g(n)` is the path cost from the start node to `n`, and `h(n)` is the estimated cost from `n` to the target.
3. **Bidirectional A* Algorithm**: Bidirectional A* runs two simultaneous A* searches—one forward from the start node, and one backward from the target. The algorithm stops when the two searches meet, potentially reducing the search space and time needed to find the shortest path.
4. **Raptor Algorithm**: Raptor (Round-based Public Transit Routing) is specialized for finding the quickest routes in public transit networks. It processes the network in rounds, considering all possible transfers from a station, and keeps track of the best routes discovered so far for efficient computation.

References and lifetimes & smart pointers

Initially, our plan was to use references extensively throughout our application. While this is an efficient way to manage data in Rust, we quickly encountered challenges with ownership and borrowing, especially in complex scenarios or concurrent contexts. These bad design decisions slowed our progress in building a solid and scalable foundation of our codebase.

To overcome these issues, we switched to Arc, a type of smart pointer in Rust that allows multiple references to the same data. By using Arc, we improved our ability to manage shared data in a multi-threaded environment.

Dependencies

- **chrono** - is a comprehensive date and time handling library. We use this crate only for local time with timezone.
- **geo** - crate provides operations like calculations of areas, lengths, and distances locations. Our usage: for finding nearest stops -> calculating distance between some point and stop using HaversineDistance.
- **gtfs-structures** - This crate simply parses GTFS files.
- **trigram** - Trigram offers a method to measure the similarity between texts by using trigram indexing, which is useful for fuzzy matching and searching within strings.
- **serde & serde_json** - Serde is a powerful serialization and deserialization framework for Rust, and serde_json is its extension for handling JSON data, enabling easy conversion between JSON and Rust data types.
- **tiny_http** - lightweight, low-level HTTP server library, designed to be simple and efficient.
- **url** - provides url parsing, we added this crate just for parsing query parameters.

Evaluation

What went well

Initially, we were unsure of the capabilities and functionalities of the gtfs-structures crate. However, upon implementation, we were pleasantly surprised by its efficiency. The crate eliminated the need for manual parsing of GTFS CSV files, which significantly streamlined our development process. This automation not only saved us time but also reduced the potential for parsing errors, allowing us to focus more on core functionalities rather than data handling.

Despite concerns about Rust's complexity - after seeing slides of our lectures on how Rust impacted development in companies - we found our development pace to be relatively quick. Engaging with Rust's borrow-checker and lifetime management indeed presented challenges, typical of Rust programming. However, these challenges were instrumental in ensuring robust and safe code. The overall experience was enjoyable.

What went not so well

After successfully testing the core functionality of our transit routing application, the last step was to wrap it in a REST API, to allow external access to our data. Initially, we planned to use the Actix web framework, a popular asynchronous framework in Rust, known for its high performance and flexibility. However, the unnecessary complex structure of our codebase presented a significant challenge when integrating with Actix.

Our primary data structure, TransitIndex, encapsulated all the information needed for routing and data processing. When building REST API endpoints, we needed to pass this structure to multiple handlers. Unfortunately, due to Rust's strict ownership and lifetime rules, we couldn't guarantee to the borrow checker that the TransitIndex would outlive all the asynchronous operations in Actix. This limitation led to significant issues with borrow-checking, preventing us from easily integrating Actix into our project.

To resolve this problem, we decided to switch to a simpler synchronous HTTP library called tiny_http. Unlike Actix, tiny_http processes requests synchronously, allowing us to avoid the complexities of asynchronous code and the associated borrow-checking challenges.