Project 1.2: Transitor

CS Year 1 Project 2

Department of Advanced Computing Sciences

Maastricht University

Table of contents

1 Transitor

In this project, "Transitor," you are tasked with developing a Java-based routing engine for Maastricht's public transport. This tool aims to analyze the socio-economic accessibility of various areas within the city. By integrating geographical data, public transport information, and socio-economic metrics, you will create a system that not only navigates through the city's complex transportation network but also sheds light on the accessibility disparities across different neighborhoods.

2 Project Description

This project primarily involves creating a Java-based navigation system for Maastricht, aimed at evaluating the socio-economic accessibility across various regions. The objective is to use this system to understand how different areas connect and relate in terms of available socio-economic opportunities.

2.1 Background

Socio-economic opportunities, including healthcare, education, and mobility, are critical for community well-being but are not uniformly accessible across urban landscapes. The distribution and accessibility of these opportunities may affect residents' quality of life and their ability to access essential services (Litman 2017). The case of Bijlmer, an area in Amsterdam, serves as an example where urban design failed to adequately provide socio-economic opportunities, leading to isolation and reduced quality of life for its residents (Olsson and Loerakker 2013).

In global contexts, studies in cities like London and New York have provided valuable insights into how transport systems affect socio-economic accessibility. For instance, London's Crossrail project was studied for its impact on regional accessibility, revealing how new transport links can transform access to jobs and services (TFL 2019). Similarly, New York's subway system analyses have demonstrated the relationship between transport accessibility and socio-economic mobility (Kaufman et al. 2014).

For Maastricht, understanding the link between transport infrastructure and socio-economic opportunity distribution is just as important. By examining this relationship, we can pinpoint

areas needing improvement for more equitable urban planning and policy-making, ultimately enhancing the city's socio-economic accessibility.

2.2 Objectives

1. Given two postal codes (zip codes), identify all feasible routes between them, including public transport services such as buses, and walking or cycling.
2. Develop a tool to quantify accessibility for all postal codes in Maastricht.

2.3 Routing Engine and Data

For this project, you will need:

- Detailed geographical information of Maastricht's Wijk (*districts*), areas (*sub-districts*), and postal codes (*specific neighborhoods*).
- Public transport information such as bus stops, routes, and travel time.
- Socio-economic opportunities within Maastricht. These encompass services including but not limited to cafes, restaurants, education, transportation, financial, healthcare, entertainment, arts & culture, public services, facilities, and waste management [1].

These will enable you to build a routing engine capable of computing different transportation options from one address to another in Maastricht. Use this engine to identify areas more isolated from specific opportunities.

3 Phases of the Project

- P1. Develop a distance and time calculator for specific locations.
- P2. Public Transport Routing (no transfers).
- P3. Public Transport Routing (with transfers) + accessibility calculations per postal code.

3.1 Phase 1: Distance and Time Calculators

In this phase, you will design a simple distance calculator between two locations.

[1] Note that this list is non-exhaustive; you may find more opportunities to include in your project.

3.1.1 Tasks:

1. Develop well-structured code to calculate the distance 'as the crow flies' between two postal codes within Maastricht, ignoring road structure and calculating aerial distance.

   Use the `MassLatLon.xlsx` file. [2] All locations are in latitude and longitude which can be used for distance/time calculations.

2. It is possible that some postal codes are not present in the file. For such cases, we have provided a web service that will allow you to look up latitude and longitude given a postal code. Select and apply a suitable design pattern for consuming a RESTful API in order to access this service.

   Use the API only if the postal code is not present in `MassLatLon.xlsx`. More information on the API is present on computerscience.dacs.unimaas.nl. [3]

3. Enforce a rate limit on API calls:

Period	Limit
5 Seconds	1 request per IP
1 Minute	5 requests per IP
1 Hour	40 requests per IP
1 Day	100 requests per IP

4. Design a simple visualization to represent distances.

   A simple plot/graph/map is sufficient at this point.

3.1.2 Requirements:

1. The distance and time must utilize latitude and longitude coordinates for its coordinate system.
2. You must calculate time for both walking and cycling.
3. You must use kilometers and minutes as standard units of measurement.
4. Provide justification for your design choices for the calculator and API interactions.

[2] You can find all the project related files on Canvas and on `https://computerscience.dacs.unimaas.nl/project12`. Please note you will need to use the UM VPN to access the project web page.

[3] You need to use the UM VPN to access this webpage.

3.1.3 To Pass:

- Have a working distance and time calculator that correctly determines the time and distance for both walking and cycling. You must write this calculator yourself and it must execute locally; you cannot use a cloud-based API or the internet for this
- Provide time and distance from DACS to Vrijthof (this should be in your slides).
- Create an elementary visualization, similar to flat public transport city maps. Include minimal components (city boundaries, Wijk), scaling for distance, and time visualization.

3.1.4 Demo:

During your first presentation, we will give you two postal codes to find the distance, calculate time, and present visualization.

3.1.5 Criteria for a higher grade (include):

1. Calculate the actual (not aerial) distance using the supplied .osm file and external libraries (e.g., GraphHopper routing engine (GmbH 2023)). Note: You must still implement your routing engine (as instructed above); using an external routing engine does not void that requirement. The external library you choose should perform routing locally; you cannot use a cloud-based API or the internet for this.
2. Develop an advanced, user-friendly graphical interface.
3. Ensure code is clean and free from common coding flaws such as bad code smells.

3.2 Phase 2: Public Transport Routing

For this phase, you have been provided with a GTFS (General Transit Feed Specification[4]) dataset for The Netherlands. Calculate the distance and time between two postal codes, now considering public transport options.

Assume that the only mode of public transport in Maastricht is by bus (ignore the Arriva train connections between Randwyck, Centraal, and Nord stations).

3.2.1 Tasks:

1. Develop an efficient algorithm to find the closest bus stops, bus routes, and times for trips for a given postal code.

[4]Please see Appendix for more information on GTFS.

2. Store the GTFS data[5] in a well-structured relational database and write efficient SQL queries.

   Hint: Use indexes where appropriate.

3. Enhance your visualization to accurately depict bus routes.

   You may use the city bus map as inspiration.

4. Write JUnit test cases and report code coverage statistics (aim for very high coverage).

3.2.2 Requirements:

1. Store GTFS data in a relational database. Optimize all SQL queries for performance (minimal query wait time).
2. Calculate time by bus. Only direct connections count. If no direct connection exists between two postal codes the GUI must indicate this.

3.2.3 To Pass:

1. Have a correctly implemented routing algorithm for finding the shortest direct route.
2. Include a GUI with all bus routes. Upon user input the GUI must indicate in some manner which bus route the user should take.

3.3 Phase 3: Public Transport Routing (with transfers) + Accessibility calculation per postal code

In this phase, focus on improving public transport routing by allowing transfers between buses, enabling riders to reach more destinations as part of their journey.

Define a measure of socio-economic accessibility[6]. Calculate the socio-economic accessibility of every postal code to identify the most and least well-connected areas of Maastricht.

[5] You can download the GTFS file from Canvas and https://computerscience.dacs.unimaas.nl/project12.

[6] This is a term in use in the field of sociology. It has several related definitions and needs to be "filled out" when considering a specific location.

3.3.1 Tasks:

1. Write an efficient routing engine that considers bus transfers.
2. Serialize and process the GeoJSON files[7] for various amenities in Maastricht.
3. Compute a detailed accessibility metric for each postal code.
4. Refactor the code, if needed, to ensure good design practices are incorporated. Provide an informal software architecture.

3.3.2 Requirements:

1. All previous requirements apply.
2. Handle the case where a postal code does not have access to a bus route, if such a case exists.
3. If the user must wait in between buses, this time must be counted as part of the travel time. See the Appendix for more information on how transit time must be calculated.

3.3.3 To Pass:

- Correct any errors from the previous phases.
- Have a functional routing engine with bus transfers. It must be integrated with the GUI.
- Efficiently calculate the accessibility measure for all postal codes in Maastricht (execution time should be as minimal as possible).
- Defend your measure of socio-economic accessibility using the provided data.

3.3.4 Criteria for a higher grade include:

- Well-structured code (no bad code smells).
- Well-designed accessibility measure, backed by scientific literature.
- Visualization of accessibility (could be separate from routing visualization).

References

GmbH, GraphHopper. 2023. "GraphHopper Routing Engine." https://github.com/graphhopper/graphhopper.

Kaufman, Sarah, Mitchell L Moss, Justin Tyndall, and Jorge Hernandez. 2014. "Mobility, Economic Opportunity and New York City Neighborhoods." *NYU Wagner Research Paper*, no. 2598566.

Litman, Todd. 2017. *Evaluating Accessibility for Transport Planning*. Victoria Transport Policy Institute Victoria, BC, Canada.

[7] See Appendix for more information on GeoJSON files.

Olsson, Lea, and Jan Loerakker. 2013. "Revisioning Amsterdam Bijlmermeer." *Failed Architecture*. https://failedarchitecture.com/the-story-behind-the-failure-revisioning-amsterdam-bijlmermeer/.

TFL. 2019. "Crossrail Project 2019 to 2023: Completing the Elizabeth Line." *Association for Project Management*. https://www.apm.org.uk/v2/media/10vnyi1q/crossrail-project-2019-to-2023-completing-the-elizabeth-line-apm-journal.pdf.

Appendix

General Transit Feed Specification (GTFS)

In phase 2 the primary data format you'll work with is the General Transit Feed Specification (GTFS). GTFS is a common format for public transportation schedules and associated geographic information. It's used worldwide to power trip planners and transit applications, making public transport data universally accessible and useful. GTFS helps various stakeholders from transit agencies to developers and passengers in planning, analyzing, and using public transport more effectively.

Understanding GTFS

GTFS consists of a series of text files (.txt) zipped into a single archive. Each file models a particular aspect of the transit system:

- **agency.txt**: Details about the transit agency, including name, URL, and time zone.
- **stops.txt**: Information about individual stops, stations, or areas where vehicles pick up or drop off passengers.
- **routes.txt**: Descriptions of routes in terms of long and short names, route type (bus, tram, etc.), and route ID linking to other files.
- **trips.txt**: Trips for each route, including trip IDs and headsigns, linked to schedules and sequences of stops.
- **stop_times.txt**: Times that a vehicle arrives at and departs from individual stops for each trip.
- **calendar.txt** or **calendar_dates.txt**: Service dates, showing when services are available or exceptions occur.
- ... and potentially more, depending on the agency's data.

GTFS data allows you to map out routes, calculate distances and travel times, and understand service patterns. In this phase, you'll develop algorithms to parse GTFS data, find the closest bus stops, and calculate the fastest routes.

GeoJSON files

GeoJSON is a standard format for encoding geographic data structures, making it an essential asset for mapping and spatial analysis. You'll use this data to identify amenities, shops, tourism spots, and other points of interest, understanding how they contribute to the socio-economic landscape of Maastricht.

- **GeoJSON Data.zip**:
  - **Tourism (tourism.geojson)**: Features 259 points of interest related to tourism like museums and historical sites.
  - **Shops (shop.geojson)**: Details about 697 shopping locations and stores, providing insights into commercial accessibility.
  - **Amenities (amenity.geojson)**: Contains 1173 features representing various public amenities and facilities.

Consider how access to different amenities, shops, and tourist attractions might influence the quality of life and opportunities available to residents and visitors alike.

Calculating transit times when bus transfers are included

In our example, at some time "start", a user is attempting to travel from alpha to beta by taking bus b1 to transfer point T, where they exit b1 and board bus b2 for the rest of the trip to beta.

Informally, the below paragraphs state: You can assume the user will take the buses and transfers that minimize the time required to get from alpha to beta.

More formally, let us assume we have a function spot(b, pc), which returns the time at which bus b is at location pc and is undefined if b is never at pc. If we stick to just those (b, pc) where spot is defined, we can now define:

- wait = spot(b1, alpha) - start where wait >= 0
- t1 = spot(b1, T) - spot(b1, alpha) where t1 > 0
- transfer = spot(b2, T) - spot(b1, T) where wait >= 0
- t2 = spot(b2, beta) - spot(b2, T) where t2 > 0

If we assume that the user will minimize wait, t1, transfer, and t2 (e.g., by not aimlessly riding the bus around all day) then we can define:

- tot = wait + t1 + transfer + t2

Note that tot is a function with 6 parameters (alpha, start, b1, T, b2, beta). In order to find the shortest travel time you must minimize tot. You will be given alpha, beta, and start, you can choose any value for the other parameters (b1, T, b2) given the restrictions described above.

Note that the term "bus" is being used in a technical sense here. For example, if the Number 6 bus stops in alpha at 9:45 and 10:12, they are considered two separate buses for the purposes of this calculation.

Start-independent total time: What if the user does not care when they leave, but just wants to know the shortest time it takes to get from alpha to beta whenever that may be?

In that case, wait is always 0 [8]. As before, you will be given alpha and beta. tot is now obtained by calculating tot over all values of (b1, T, b2) and taking the minimum result.

[8] This can be best understood as setting start == spot(b1, alpha) in the appropriate equation