The Impact of an Extra Feature on the Scalability of Linked Connections

Pieter Colpaert, Sander Ballieu, Ruben Verborgh, and Erik Mannens

Ghent University – iMinds – Data Science Lab
firstname.lastname@ugent.be

Abstract. Calculating a public transit route involves taking into account user preferences: e.g., one might prefer trams over buses, one might prefer a slight detour to pass by their favorite coffee bar or one might only be interested in wheelchair-accessible journeys. Traditional route planning interfaces do not expose enough features for these kind of questions to be answered. In previous work, we proposed a Linked Data interface, called Linked Connections, which allows user-agents to evaluate the route planning queries on the client-side, and thus allow for extra features to be implemented by data reusers. In this work, we study how and where these new features can be added to the Linked Connections framework. We researched this by adding the feature of wheelchair accessibility both on server and client, and comparing these two solution on query execution time, cache performance and CPU usage on server and client. We found that for the use case of wheelchair accessibility, there is no advantage of adding this feature on the server: the query execution time does not improve, while the cache hit rate lowers.

Keywords: Linked Connections, route planning, Connection Scan Algorithm, wheelchair accessibility, public transit

1 Introduction

It is in the interest of transport companies to maximize the reuse of their data. Having the latest updates on every possible screen will indeed give a higher customer satisfaction to the traveler. Today, we see evidence thereof by public transit companies that publish their data as data dumps on the Web using the General Transit Feed Specification (GTFS) on the one hand, or expose route planning interfaces over HTTP on the other. One might want to have the fastest route while a passenger with heavy luggage wants to minimize the number of transfers. Other travelers want to find journeys that are not only short but also pleasant [5] or journeys that are accessible with a certain disability. To this end, Linked Connections [2] was introduced as a way for user-agents to download the data just in time, when executing the route planning algorithm itself.

In this paper, we investigate how such extra features can be added to a transit route planner that uses the Linked Connections framework. More specifically, we add the possibility to filter on journeys that are wheelchair-accessible, as this is a good example of a personal feature which the client can know in advance. We wonder what the effect on the query execution time, cache hit rate and CPU time is when adding a wheelchair accessibility filter on the server, compared to adding nothing to the server at all. Our hypothesis is that adding a filter on the server-side will take off load from the client and add load to the server. Furthermore, when adding a filter on the server-side and under "normal" load, we expect to see a lower query execution time.

In the next section the background and related work within public transit route planning are described. Then the Linked Connections framework is introduced followed by the changes that are needed to support wheelchair-accessible journeys. We then describe the used evaluation method and the corresponding results. Finally, we conclude and give perspectives for consuming and publishing Linked Data.

2 Related Work

GTFS, as used by Google Maps since 2005, contains a collection of text files. It defines the base vocabulary we are working with:

stop A location where passengers board or disembark from a transit vehicle[1].

trip A collection of stops followed by a transit vehicle at specific times[2].

route An advertised route followed by one or more trips that follow a similar set of stops[3].

transfer A rule to transfer from one stop to another[4].

The most relevant files within GTFS are: *stops.txt*, *routes.txt*, *trips.txt*, *stop_times.txt*, and *transfers.txt*. GTFS is also available as Linked-GTFS at *http://vocab.gtfs.org/*, which is a mapping of these terms to URIs. However, the de facto way to exchange transit data is to send this file over e-mail or publish a zip-file on the Internet using FTP or HTTP.

The problem that we need to solve using this data is the Earliest Arrival Time (EAT) problem. An EAT query consists of a departure stop, a departure time, and a destination stop. The goal is to find the fastest journey to the destination stop ($q_{deststop}$) starting from the departure stop ($q_{depstop}$) at the departure time ($q_{deptime}$). In order to solve EAT queries, the Connection Scan Algorithm (CSA) [3], introduced in 2013, uses a stream of connections. A *connection* is a combination of a *departure stop* ($c_{depstop}$) with a *departure time* ($c_{deptime}$) and an *arrival stop* ($c_{arrstop}$) with an *arrival time* ($c_{arrtime}$). It represents a public transit vehicle that goes from $c_{depstop}$ at $c_{deptime}$ to $c_{arrstop}$ at $c_{arrtime}$ without stopping at an intermediate stop. All connections are combined into a connection stream, sorted by increasing departure time. The CSA algorithm scans every reachable connection. Because the stream of connections is sorted by departure

[1] *http://vocab.gtfs.org/terms#Stop*
[2] *http://vocab.gtfs.org/terms#Trip*
[3] *http://vocab.gtfs.org/terms#Route*
[4] *http://vocab.gtfs.org/terms#TransferRule*

time, it is sufficient to only consider the connections where the $c_{deptime}$ is later than $q_{deptime}$. A connection is *reachable* when there exists a series of connections that starts at $q_{depstop}$ and ends at $c_{depstop}$. When a new scanned connection leads to a faster route to the arrival stop, the MST will be updated. This is the case when $c_{arrtime}$ is earlier than the actual EAT at $c_{arrstop}$. Finally the algorithm ends when the destination stop is added to the minimum spanning tree. The resulting journey can be obtained by following the path in the MST backwards, starting from $q_{deststop}$ to $q_{depstop}$.

Linked Connections (LC) is a framework for publishing public transit data [2]. The framework publishes and consumes data using the Linked Data vocabularies of Linked-GTFS, Linked Connections and Hydra. It consists of a server which is responsible for publishing connections in an ordered fashion and a client that uses the Connection Scan Algorithm to process these connections. The LC server publishes its connections as Linked Connections Fragments (LCF) over HTTP using a REST API. A LCF is a part of a list of connections that is sorted by departure time and then fragmented based on departure time intervals. The different LCFs are connected with each other by hypermedia-links: it contains a link to the next and the previous page. This allows the server to dynamically shrink or expand the departure time intervals depending on the number of connections within the interval. The Hydra ontology [4] is used to specify the next and previous page links as well as how the resource itself should be discovered.

The LC client is responsible for building a stream of connections that can be given to the Connection Scan Algorithm as input. In order to do this the LC client downloads LC fragments from the LC server. The URI template to discover the first page is specified in the entry point of a LC server. To get the following LC fragments the client needs to follow next page links. Because it is likely that multiple identical HTTP requests will be executed by the same or other user-agents, both server and client caching are enabled.

3 Linked Connections with wheelchair accessibility

A wheelchair-accessible journey has two important requirements: first, all vehicles used for the route should be wheelchair-accessible. Thus, the trip (a sequence of stops that are served by the same vehicle), should have a wheelchair-accessible flag set to true when there is room to host a wheelchair on board. Secondly, every transfer stop, the stop where a person needs to change from one vehicle to another vehicle, should be adapted for people with limited mobility. As the LC server does not know where the traveler is going to hop on as it only publishes the time tables in pages, it will not be able to filter on wheelchair-accessible stops.

When extending the framework of Linked Connections with a wheelchair accessibility feature, there are two possible ways of implementing this:

1. Filter both the wheelchair-accessible trips and stops on the client, and
2. Only filter the wheelchair-accessible trips on the server, filter the stops on the client.

3.1 Linked Connections with filtering on the client

In a first approach, the server does not expose wheelchair accessibility information and only exposes the Linked Data of the train schedules using the Linked Connections vocabulary. The client still calculates wheelchair-accessible journeys on the basis of its own sources. We thus do not extend the server: the same server interface is used as in LC without wheelchair accessibility. The LC client however is extended with two filter steps: the first filter removes all connections from the connections stream whose trip is not wheelchair-accessible. The second filter is added to CSA, to filter the transfers stops. When CSA adds a new connection to the minimum spanning tree (MST), it detects whether this would lead to a transfer. CSA can use the information from stops in the Linked Open Data (LOD) cloud to decide if the transfer can be made and the connection can be added to the MST. To be able to support dynamic transfers times, CSA can also request Linked Data on "transfers" from another source. When a transfer is detected, the Connection Scan Algorithm will add the transfer time to the departure time of the connection. The trips, stops and transfers in our implementation are simple JSON-LD documents. They are connected to a module called the *data fetcher* that takes care of the caching and pre-fetching of the data.

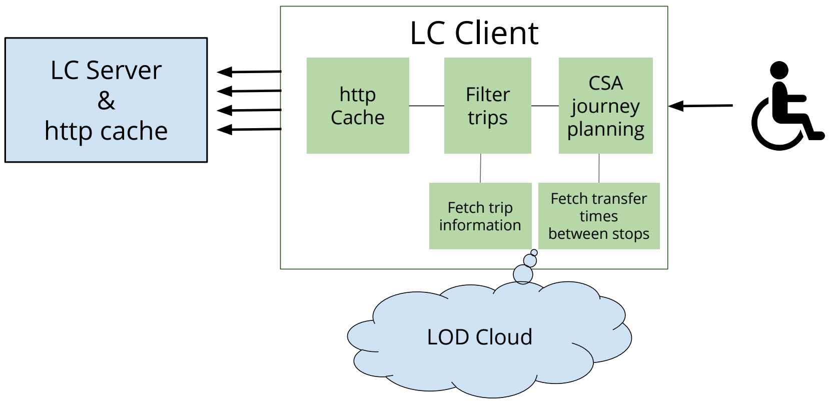

Fig. 1. Architecture for Linked Connections with wheelchair accessibility filtering on the client-side.A user that needs wheelchair-accessible route planning advice consults the LC client. The LC client discovers the first page of a connections stream through the entrypoint of a LC server. Once the connections stream is set up by downloading the next page linked from every HTTP response, the LC client can filter the wheelchair-accessible trips based on data it got from the LOD cloud. The filtered connections stream is then provided to the CSA algorithm, which when discovering transfer times also takes into account the wheelchair accessibility of a stop. Every HTTP request is easily cacheable as each client will request similar pages.

This solution provides an example of how the client can now calculate routes using more data than the data published by the transit companies. As wheelchair accessibility is specified in the GTFS standard, we can expect that the public transit companies provide this data. In practice we have noticed that the data is mostly not available, which can be consider normal for an optional field. This makes us believe an external organisation that represents the interests of the less mobile people should be able publish this data.

3.2 Linked Connections with filtering on the server and client

In the Linked Connections solution with filtering on both server and client-side, the trips filtering, as illustrated in Figure 2. The server still publishes the time schedules as Linked Data, yet an extra hypermedia control is added to the LC server to enable the trips filter on the resulting connections. The wheelchair accessibility information is directly added to the servers' database, and an index is configured, so that the LC server can query for this when asked.

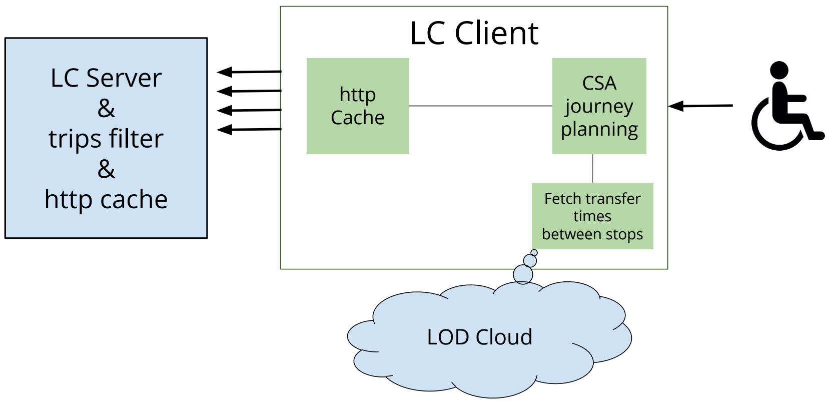

Fig. 2. Architecture for Linked Connections with wheelchair-accessible trips filtering on the server-side shares a similar narrative with Figure 1. Now, the server however already knows the wheelchair accessibility of the trips and exposes a filter functionality over HTTP.

4 Evaluation

The purpose of the evaluation is to compare the two LC solutions based on the scalability of the server-interface, the query execution time of an EAT query and the CPU time used by the client.

4.1 Query mixes

There exists an Open Data source for the query logs of a real route planner in Belgium [1] called iRail[5]. As we also have access to the timetables of the Belgian

[5] *http://api.irail.be*

railways, we chose to benchmark a route planner for Belgium. From these logs we made approximate query mixes where the average number of queries per second grows linearly, by taking the 15 minutes during peak hours on the first of October 2015, and consequentially adding the next 15 minutes as if they also happened during that time[6]. Figure 3 plots the average number of queries per second for the 10 query mixes.

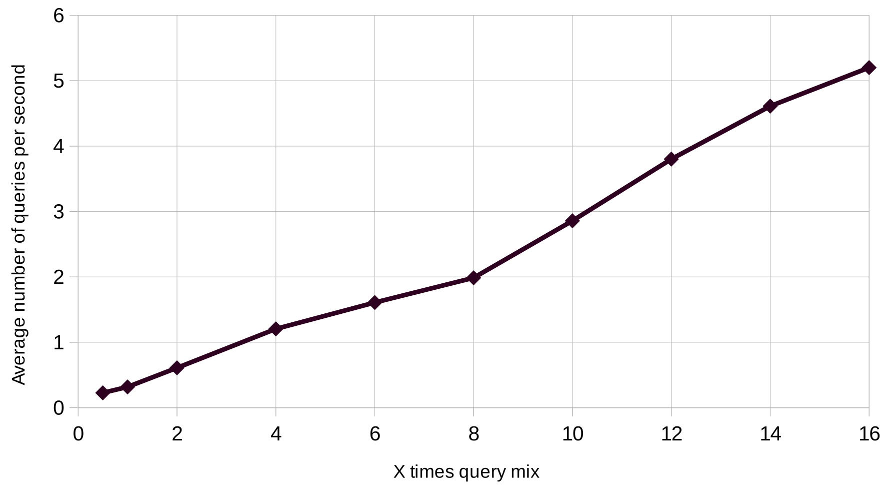

Fig. 3. The 10 query mixes that are used for the evaluation have a growing average number of queries per second.

The iRail route planner does not support wheelchair-accessible route planning and thus the logs do not contain any information about wheelchair accessibility. According to European statistics[7], 13.9% of Belgians have a basic activity difficulty in 2011. Not all of these need wheelchair-accessible journeys, yet also people pushing strollers might be interested in wheelchair-accessible journeys. For the query mixes, we therefore randomly added a wheelchair accessibility flag in 10% of the queries.

4.2 Evaluation design

Following resources have been set up for the evaluation:

- *Two LC servers* one for the LC with filtering on the client setup and one for the LC with trips filtering on the server setup. The MongoDBs used by the

[6] You can download these query mixes at *https://github.com/linkedconnections/benchmark-belgianrail*

[7] *http://ec.europa.eu/eurostat/statistics-explained/index.php/Disability_statistics_-_prevalence_and_demographics*

LC servers are populated with connections of the Belgian railway company of 2015.
– A *transfers, trips and stops resource* as a data source for the wheelchair accessibility information.

A NGINX proxy cache server is installed in front of each HTTP server to enable compression and caching with a max memory of 1GB. Each Linked Connections resource is configured to be able to cached for 1 minute, which is the update velocity of the information of the Belgian railway company. The other resources are fetched once when the client is started.

On each run, a specific query load and route planner is selected. Then, the NGINX cache is cleared for all HTTP servers. During a period of 15 minutes the queries from the query mix are sent to the route planner depending on their time offset and four metrics are measured during this evaluation:

– *The percentage CPU usage at the server side* measured by measuring the CPU time used by the server application and divide it by the CPU time already passed since the start of the application. The *pidstat* command from the *Sysstat* package was used to calculate the CPU time.
– *The percentage CPU usage at the client side* measured the same way, now on the client application.
– *The query execution time per connection.* As the time complexity of the query execution time is $O(n)$ with n the number of connections, we divide the execution time by the number of scanned connection to have a fair comparison.
– *Cache hit rate at the client-side* by counting the number of cache misses and divide it by the number of hits and misses.

We only take into account one agent that has to process multiple queries. Multiple agents could nonetheless be simulated by disabling the client-side cache. The results would be similar, only the server cache would be used instead of the client cache.

The experiments were executed on an Intel(R) Core(TM) i5-3470 CPU @ 3.200 MHZ with 8 GB of RAM with each script being executed on a single thread.

5 Results

Figure 4 contains the CPU load of the server. The server interface with filtering on the server has a similar scalability as without the filter functionality. However, an average raise of 1.24% in processing power needed can be noticed.

In Figure 5, we can see the CPU load of the client performing the algorithm. When the query load is half of the real iRail load, we notice that the load is 20% for filtering on the client and 21% for trips foltering on the server. Continuously increasing the query load results in an increased client CPU load. The client load of the solution with filtering on the client increases from 27% at query mix 1 to 93% at query-mix 16 while the solution with filtering on the server-side increases from respectively 30% to 93%. For query loads higher than 12 times the iRail load the difference between the two solution becomes less than 1%.

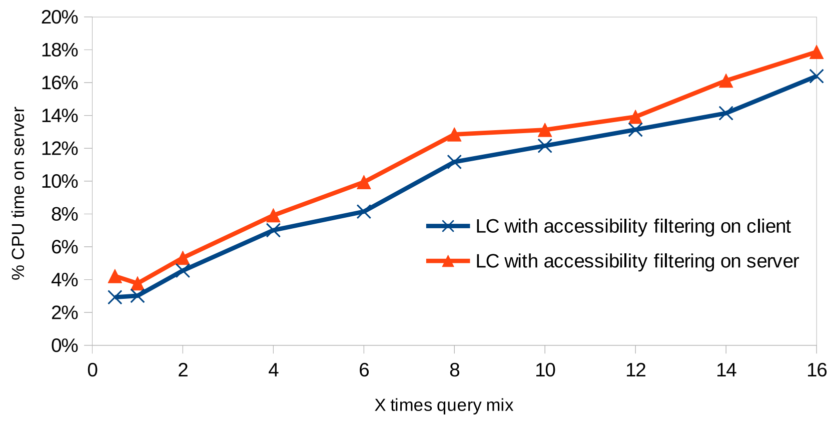

Fig. 4. The CPU usage at the server-side under increasing query load shows that wheelchair accessibility filtering on the server takes more effort for the server under all query loads.

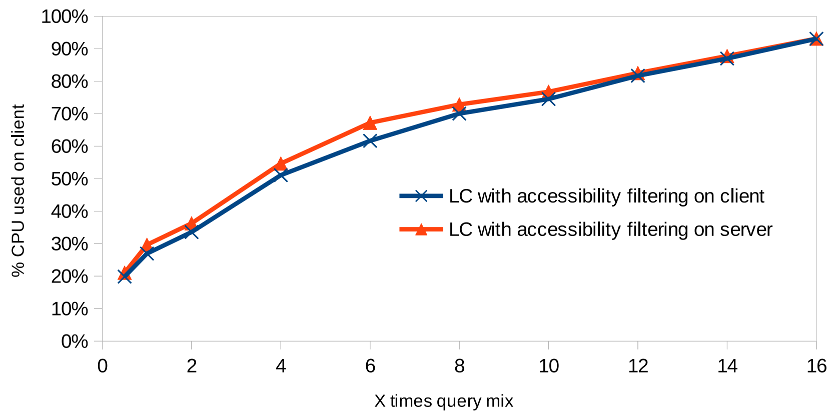

Fig. 5. The CPU usage of the client is higher under increasing query load for the LC setup with filtering on both the client and server-side than only on the client-side.

Figure 6 shows the average query response time per connection. We notice that the fastest solution is the solution with filtering on the client for lower query loads, yet for higher query loads, the execution times become comparable.

The cache performance for the three evaluated setups is given in Figure 7. When observing half of the normal iRail query load the measured cache rate is 76% and 70% for respectively with filtering on the client as with trip filtering on the server. The cache hit rate slowly decreases to respectively 70% and 64% at query mix 8 and finally to 64% and 61% at the highest query mix 16. When observing all query loads, the cache hit rate measured from the Linked

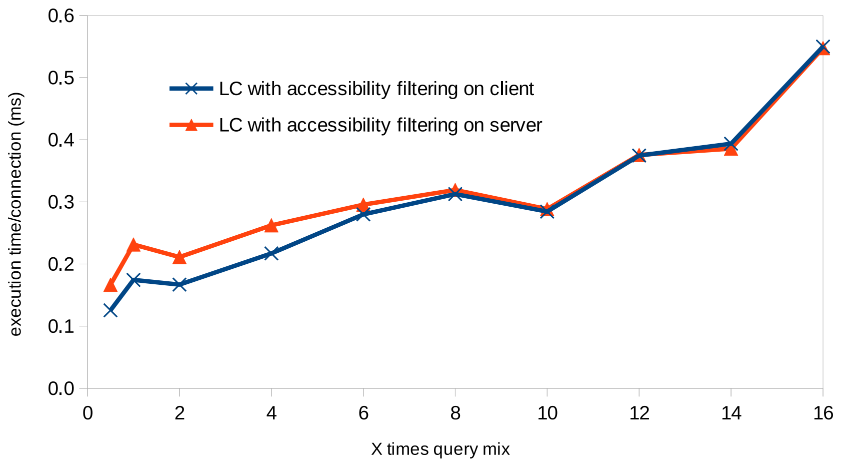

Fig. 6. The average time needed to process one connection in milliseconds shows that for the normal query loads the Linked Connections solution with trip filtering on the server is on average slower than the solution with filtering only on the client. When the query load is 8 times the normal query load, we can no longer notice a difference.

Connections framework with filtering on the client is lower than the framework with filtering on both client and server-side.

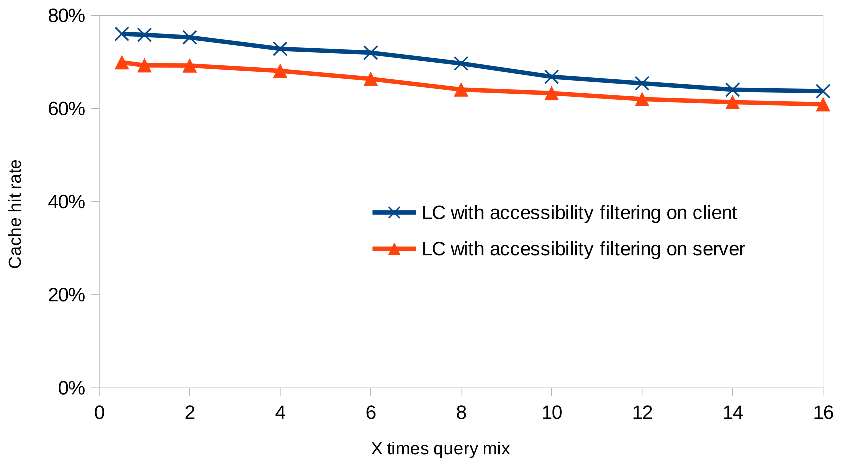

Fig. 7. The cache hit rate shows that a boolean filter on the server reduces the cache performance by 3% to 6%

When comparing the two LC implementations we can observe that moving the trips filter from the client to the server-side does not cause an improvement of the query execution time. The cache performance is better for the client-filting solution because hybrid solution needs an extra parameter to query the LC server. This results in more unique requests and consequently in a lower cache hit rate. The loss of cache performance increases the number of requests that the LC server needs to handle which leads to a higher CPU load at the server-side and an increased query execution time. The difference becomes smaller as the query load increases, as more queries can use the already cached Linked Connections documents.

6 Conclusion

Our hypothesis that the server can help user-agents by doing the filtering on trips, did not turn out to be true. There is no real advantage to be found for adding a boolean filter on the server for wheelchair accessibility as the query execution time does not improve. Furthermore, the server needs to invest 1.2% more CPU time in order to answer the same queries, and the client needs to invest between 0.1% and 5.6% more CPU time. These results can be attributed to the drop in cacheability of filtering trips on the server. For the Linked Connections framework, we will not add a feature to the data publisher for filtering on wheelchair-accessible trips. Instead, the distributed nature of Linked Data allows anyone to publish wheelchair-accessible trips and stops in a different resource, which can also be discovered and consumed by user-agents.

Finally there are also other aspects to take into account which can lead to an increased user experience when using the option with all filtering on the client. As the filtering is executed by the client, we can easily change the parameters that are used within the filter. For example the transfer times can be changed based on your own walking speed, or you can choose to use an external source (different from the public transit company) to determine the wheelchair accessibility of the stops. Running the algorithm on the client-side also insures the privacy of the end-user: departure stops and arrival stop are never sent to the public transit agencies.

For consuming and publishing Linked Data on the Web using public interfaces, we might never know at the time of publishing what consumers are going to be doing with the data. Adding functionality on the server might not always lead to a better Linked Data reuse framework, as evidenced by this paper.

References

1. P. Colpaert, A. Chua, R. Verborgh, E. Mannens, R. Van de Walle, and A. Vande Moere. What public transit API logs tell us about travel flows. In *Proceedings of the 6th USEWOD Workshop on Usage Analysis and the Web of Data*, Apr. 2016.
2. P. Colpaert, A. Llaves, R. Verborgh, O. Corcho, E. Mannens, and R. Van de Walle. Intermodal public transit routing using Linked Connections. In *Proceedings of the 14th International Semantic Web Conference: Posters and Demos*, Oct. 2015.

3. J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner. Intriguingly Simple and Fast Transit Routing. In *Experimental Algorithms*, pages 43–54. Springer, 2013.
4. M. Lanthaler and C. Gütl. Hydra: A vocabulary for hypermedia-driven web apis. *LDOW*, 996, 2013.
5. D. Quercia, R. Schifanella, and L. M. Aiello. The shortest path to happiness: Recommending beautiful, quiet, and happy routes in the city. In *Proceedings of the 25th ACM Conference on Hypertext and Social Media*, HT '14, pages 116–125, New York, NY, USA, 2014. ACM.