Master Thesis

Ontology-Based Route Queries with Time Windows

Tobias Faaß

08.06.2015

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik

Bearbeitungszeitraum
08. 12. 2014 – 08. 06. 2015

Gutachter/in
Dr. Sabine Storandt
Prof. Dr. Christian Schindelhauer

Betreuer/in
Dr. Sabine Storandt

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, 8. Juni 2015

Contents

Abstract

Most of the time we use route planning to get from A to B. However, we can also incorporate stopovers into our route so that we may get things done along the way. A route from A to B via a gas station, for example, is of no use if the gas station is closed upon arrival. As a consequence, the opening hours and the duration of stay become necessary considerations.

In this thesis we define an ontology to be a hierarchical structure of groups to serve us as stopovers. We fill this ontology with POI extracted from the open source project OpenStreetMap. Since the extracted POI rarely have attached opening hours, these opening hours must be calculated based on existing ones or defined by hand. To calculate the route along which a sequence of POI groups are visited, we implement known approaches of sequenced route planning and extended them to match the time window requirements.

Different test queries were executed on the system. Within a test area of about 25 km, query times under a half second were measured. During these tests, it was discovered that the route starting time influences the running time, which is due to the fact that during the night most POI are closed.

The implemented system covering all of Germany can be found at
http://panarea.informatik.uni-freiburg.de/osm-search/.

Zusammenfassung

Meistens wird Routenplanung dazu verwendet, um von A nach B zu kommen. Wir können aber auch Zwischenstopps auf unsere Route mit einbeziehen, um Dinge auf dem Weg zu erledigen. Eine Route von A nach B über eine Tankstelle hat keinen Nutzen, wenn die Tankstelle beim Eintreffen geschlossen hat. Dies macht die Berücksichtigung der Öffnungszeiten und der Aufenthaltsdauer notwendig.

In dieser Thesis definieren wir eine Ontologie, eine hierarchische Struktur von Gruppen, die uns als Zwischenstopps dienen. Wir extrahieren aus dem Open Source Projekt "OpenStreetMap" Orte von Interesse (POI) und fügen diese der Ontologie bei. Da die extrahierten POI selten beigefügte Öffnungszeiten enthalten, müssen diese auf Basis von existierenden berechnet oder händisch definiert werden. Um die Route, welche eine Abfolge von POI Gruppen besucht, zu berechnen, werden bekannte Methoden der sequenzierten Routenplanung implementiert. Diese werden so erweitert, dass sie den Voraussetzungen der Zeitfenster entsprechen.

Verschiedene Testanfragen an das System werden ausgewertet, wobei in einem Testbereich von ungefähr 25 km Abfragezeiten von unter einer halben Sekunde gemessen wurden. Während diesen Tests wurde ein Einfluss der Startzeit auf die Laufzeit entdeckt, was auf die geringe Anzahl an geöffneten POI in der Nacht zurückzuführen ist.

Das implementierte System, ganz Deutschland umfassend, ist unter der Adresse http://panarea.informatik.uni-freiburg.de/osm-search/ zu finden.

1. Introduction

We all have favourite supermarkets we usually go to. We know the paths to them and back. Perhaps we found a shortcut to them after a couple of trips. But if we have to handle multiple Points of Interest (POI) or we are in an unknown city, we can't just rely on our experience. GPS-navigation systems or web applications like Falk.de and Google Maps provide us with the information as which path to take to get to a desired location or allow us to explore a map to find the next restaurant or supermarket. Sometimes we need to visit different POI on the way to a destination like in this example: Imagine you are invited to a birthday barbecue party. You will need to buy some steaks and some flowers as a present and because you are short on cash you need to get some bills from a bank. However, you can't just go to any bank. You will need to go to a bank at which you are an account holder, for example"Kreissparkasse". Since you are short on time as well you will need to visit these places as fast as possible. Even if you would know the best route to the next bank, butcher and florist, one of them being closed on arrival would make you rethink your route.

Figure 1.1.: Route example: From the source to a bank of the operator "Kreissparkasse", followed by a florist and butcher then heading to the target destination.

We see that not only is the geo-data information of POI important, but also their temporal constraints. Efficient route planning algorithms are available for route queries via POI (Section 1.2), but none of these consider the POI opening hours. An important step towards the goal is also a wide and deep ontology providing interesting and useful groups to route via that consider not just groups like "Supermarket" but also more detailed subgroups like "Discounter" and its biggest operators like "ALDI" or "LIDL" . This gives the user the possibility of planning a route via the POI that is most suitable to the user's needs.

These queries should be answered efficiently on country-sized road networks.

To find out if real-time answering is possible we first have to create an ontology and handle opening hours as described in Chapter 2. This will make sure we will find our particular bank subgroup ("Kreissparkasse") as well as a butcher and florist. A summary of a previous project will provide a first overview of the topic. Chapter 3 will show the beginning of our route planning based on the Dijkstra algorithm and the enhancements that have been performed on it to solve the whole request. To get a picture of the whole system Chapter 4 will show the system's architecture and the web client's view. The performance of the system will be then tested in Chapter 5 followed by a glimpse into future improvements and extensions in Chapter 6.

1.1. Problem Definition and Goals

The formal definition of the routing problem considered in this thesis can be phrased as follows:

Given is a road network G(V,E), where on each Vertex V multiple POI can be located. Each POI with its opening hours and type. And an ontology containing POI assigned to hierarchical Groups according to their type. Additionally there is a query consists of a source and target vertex, the time of departure as well as optional ontology groups with their corresponding duration. The goal is to find the fastest path from source to target which visits a POI of each requested group in the selected order while considering their opening hours and duration of stay, efficiency. This problem is related to the NP-hard travelling salesman problem (TSPTW) with time windows. The problem is defined as finding a minimum-cost path visiting a set of cities exactly once, where each city must be visited within a given time window [DGLT12].

This goal was achieved by:

- Creating an own ontology that is usable in the routing context (Section 2.2).
- Implementing two known approaches for sequenced route planning: bounding box (Section 3.3) and iterative doubling (Section 3.4) and extending them with time window capability.
- Handling the POI which lack attached opening hours by an own approach of using core hours (Section 2.3).
- Creating an easy to use user-interface (Section 4.2).
- Testing the efficiency of the created system (Chapter 5).

1.2. Related Work

In the paper "DO-ROAM: Activity-Oriented Search and Navigation with OpenStreetMap" [CHK+11] a web based system called DO-ROAM for activity oriented search was introduced which primarily focused not on locations but rather on activities. An Ontology of activities (OSMonto)was created based on the Web Ontology Language OWL and implemented the DO-ROAM system in RubyOnRails and SQL. Additionally, the DO-ROAM system as described in their paper " Ontology-based Route Planing for OpenStreetMap" [MCM12] was later enhanced with a route planner which uses engines available as web services like OSRM and YOURS. The DO-ROAM system is no longer online.

In another paper " Sequenced Route Queries: Getting Things Done on the Way Back Home" [EF12] Jochen Eisner and Stefan Funke improved Dijkstra-based sequenced route planning with two efficient changes, namely Iterative Doubling, as discussed in Section 3.4, and Contraction Hierarchies [GSSD08]. They tested their implementation and showed that sequenced route planing can be done quickly and efficiently compared to the naive approach by using Iterative Doubling and Contraction Hierarchies. Instead of an ontology, their tests involved a list of only 17 different POI types like parking lots, restaurant and supermarkets. They also tested the influence of the order of POI but were surprised by the fact that the ratio between the slowest and fastest route type was only about 1.28 [EF12].

Both, the DO-ROAM and the Sequenced Route Queries paper disregard the opening hours.

2. Basics

In this chapter we explain how to retrieve an ontology from Open Street Map data and describe how to extract and extrapolate opening hours of POI.

2.1. Open Street Map

The data required to create an ontology and fill it with entities can be found in the OpenStreetMap Projekt. It is an OpenSource Project following the "Wikipedia"-principle of user-generated and managed content. People can easily adding new data by either walking around with GPS-tracking devices and add the position afterwards or simply by editing the data in the project page. This data can be extracted from the project page or various other websites like geofabrik[1] at which the OSM-data of counties or specific regions can be downloaded.

What does an OSM-file look like?

The original OSM-file is XML-based and consists of three major components:

- **Nodes**: a certain point
- **Ways**: a line of points, also used to form areas
- **Relations**: defined relations between nodes, ways or other relations

These components posses attributes such as the position in the world, represented as latitude and longitude, the date of creation, the name of the user who most recently edited the entry, as well as various tags that define what a component should be seen as. These tags define whether a way should be seen as a street, a river or a boundary of a building. They are always key-value pairs like key:amenity and value:restaurant or key:cuisine and value:italian [Ope15b].

To speed up the data access, binary formats like PBF (Protocolbuffer Binary Format) are used. The reference implementation of a reader using the PBF format is Osmosis, which is also used in this project[Ope15b].

[1] http://download.geofabrik.de

2.2. Ontology

An ontology is a definition of categories which shows the interrelationship between its entities. OpenStreetMap [Ope15a] tags have been grouped and sorted to create an ontology which not only covers a wide range of themes, but is also capable of providing good route points for queries.

Defining an Ontology

The previously mentioned tags belonging to the entities in the OSM-file can be mapped to groups in a hierarchical representation. To represent this mapping, an XML-file has been created, which consists of "groups" representing a topological group and "elements" representing a type of key-value-pair required to be a member of this group. A group can contain several elements which are interpreted as: "one of these elements have to be contained in the entity to be assigned to this group". Groups may also contain groups themselves, thus creating subgroups.

For example, a major group containing the element "amenity=restaurant" and a subgroup containing the element "cuisine=italian" would symbolize a group "Restaurant" with a subgroup "Italian Restaurants"'. The diagram below shows a navigation through the ontology from the root to the group "Discounter" which is a subgroup of "Supermarket" which is itself a subgroup of "shopping".

Figure 2.1.: Navigating through the Ontology. Shopping -> Supermarket -> Discounter

According to the OSM specifications chains like MCDonal's should be tagged in the OSM-file with "operator=MCDonald's", but usually they are not. This requires parsing of the name to sort out the entities as well. To also handle the various typing errors occurring in most user-provided content regular expressions are used. These regular expressions are tested during the OSM-file parsing process. Parts of the elements of the "Döner"-group (Food→FastFood→Döner) look like these:

```
...value=".*(D|d)(Ö|ö|((O|o)(E|e)))(N|n)(E|e)(R|r).*" key="REGEX"...
...value=".*(K|k)(E|e)(B|b)(A|a)(P|p|B|b).*" key="REGEX"...
```

The code above shows the handling of names containing Döner, doener or DÖNER as well as Kebabs and Kebaps.

The following snippet of Ontology.xml shows the definition of the restaurant and Italian restaurant groups. The icon attribute defines the name of the icon used in the user interface.

```
<group name="Food" icon="restaurant">
  <group name="Restaurant" icon="restaurant" corehours="Mo-Fr 11:00-14:00, ...
    <element value="restaurant" key="amenity" ></element>
    <element value="restaurant_und_hotel" key="tourism"></element>
    <group name="Italian Restaurant" icon="restaurant_italian">
      <element value="italian" key="cuisine"></element>
      <element value="pizza" key="cuisine"></element>
```

After parsing, these nodes will be saved in a node list while the assigned node-IDs are stored in group objects.

Sizes and Frequencies

The OSM-Files of Germany and Baden Württemberg, which are used for experiments in this thesis, have the following sizes:

	Germany	Baden Württemberg
Nodes	28.338.966	4.263.924
Ways	181.578.930	26.495.775
Relations	416.666	62.575
Extracted POIs	549.429	80.458

Figure 2.2.: OSM-file size and usage

Most of the ways, nodes and relations saved in an OSM-file are vertices of the road network or polygons of a building or an area. Only a few of these contain informations useful for the ontology. The following figure shows the frequencies of some groups extracted from the Germany OSM-file.

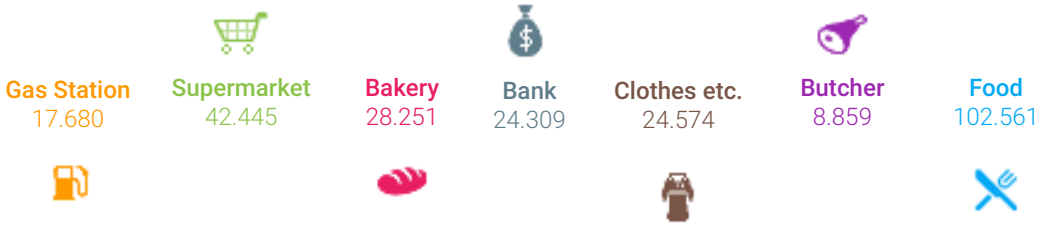


Figure 2.3.: Frequencies of different groups

2.3. Opening Hours

The optional opening_hours tag in most of the parsed POI is used to route via places which are open on arrival as well as during the given duration of stay. Valid opening_hours tags look like:

- opening_hours=24/7
- opening_hours=Mo-Sa 10:00-20:00; Tu 10:00-14:00
- opening_hours=Mo-Su 08:00-18:00; Apr 10-15 off; Jun 08:00-14:00; Aug off; Dec 25 off

The OpeningHoursParser of the OsmAnd-project[2] interprets the opening hours and creates an OpeningHours Object which can be used to get the opening status the POI within a specific time window.

Core hours

The following figure shows the percentage of parse-able opening hours attached to the POI of the main groups using the Baden-Württemberg data set.

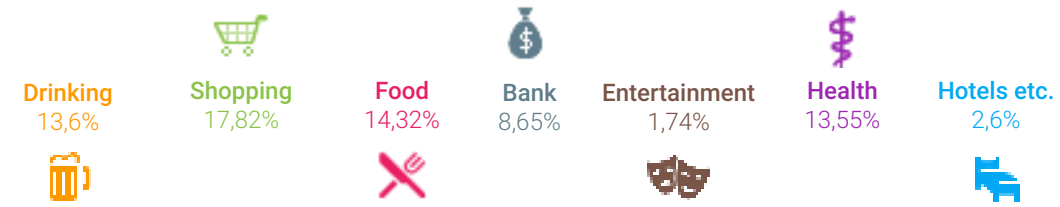


Figure 2.4.: percentage of parse-able opening hours per group

[2]http://osmand.net

Since only 28% of the "Supermarket" nodes and 11% of the "Butcher" nodes have valid opening hours attached (according to the used Baden-Württemberg data set) most of the nodes would be ignored upon the routing calculation. A calculation in which only one fourth of the group's POI are included does not provide a satisfactory routing result. We introduce the concept of core hours to increase the coverage of valid opening hours.

Core hours of a group, which represent the time period in which all POI of a group should be open, increase the coverage and improve the results. These core hours can either be initialised with a parameter in the ontology like "Mo-Fr 8:00-18:00" or calculated on server start from the other POI in the used group. The calculated core hours are simple opening hours without special rules like "week 2-52/2" or "Dec 25 off".

Calculating the core hours

To calculate these core hours a 7x24 boolean array, symbolizing the 24 hours of each day of a week, is initialized with true. Now each POI of the group with valid opening hours is tested for each hour "Monday 0:00, Monday 1:00,...". As soon as it becomes closed during this time window, the boolean variable is set to false. After testing each POI, the parts of the array which are still true symbolize the core hours. A String will be generated for the opening hours of each day ("Mo 08:00-18:00, Tu 08:00-18:00,...". If consecutive days like "Mo 08:00-18:00" and "Tu 08:00-18:00" have the same opening hours, they will be summed up to "Mo-Tu 08:00-18:00". This won't result in accurate core hours because they don't consider public holidays or rules like "Dec 25 off". The calculated core hours are just useful at different shops of an operator or POI which have similar opening hours. One bad example is the restaurant group in which each restaurant has its own days off rule. This will result in just 2 open days which therefore is useless. Having a look at the calculated core hours of a group and adjusting them is still required.

Groups like Banks, Theatres and Hospitals were initialized with core hours of 24/7 because their ATMs are usually accessible even if the bank is closed or in case of the Theatre group the opening hours depend on the currently available plays (core hours of 24/7 will make the Theatre group at least usable in the routing context). A pop-up information about calculated opening hours is shown to the user if core hours were used to solve the current request.

In the snippet of the Ontology.xml mentioned in the previous section, the attribute core hours defines the core hours that should be used if none are attached to the entity. If the attribute is true, then core hours will be calculated for this group according to its members.

2.4. Previous Work: Ontology-Search

During a university project a simple version of the ontology and the ontology map has been implemented. This version was used as a basis for the thesis and has been improved during the development as well.

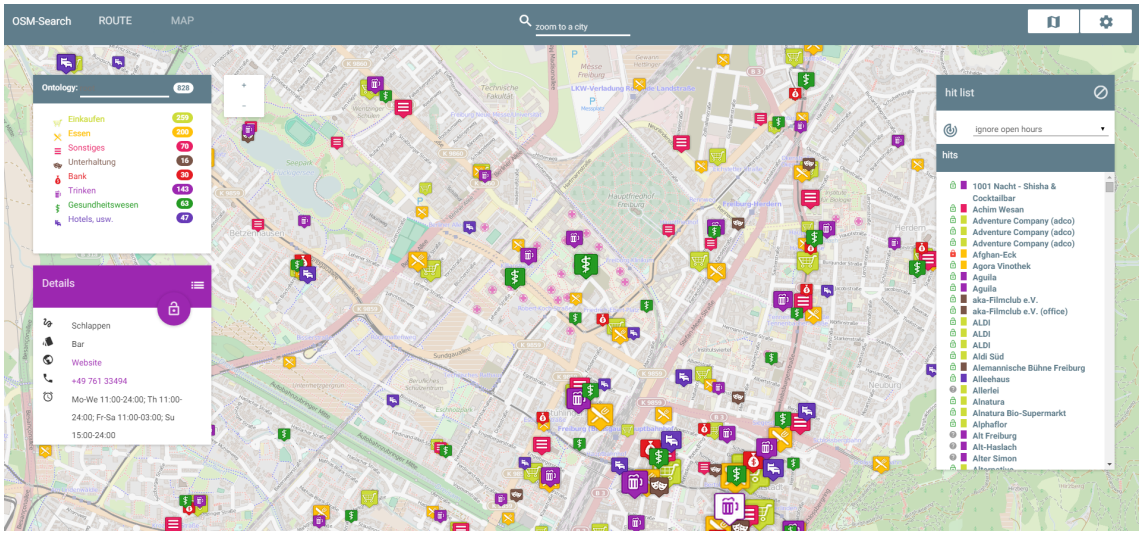


Figure 2.5.: Map overview

Ontology-Search offers browsing through the ontology (see Section 2.2) and visualizes the results in an intuitive and easy way. The results are shown as coloured and themed markers on the map enable the user to distinguish between each group easily. Based on the current zoom level, markers close to one another are clustered together to ensure a clear view. Popups on mouse overs will show the name of the POI or the cluster size and by selecting the marker, detailed information will be shown in a specific window on the left side. A list on the right side will show all hits sorted by name giving also feedback if the POI is currently open or closed.

On the server side, each group of the ontology contains a number of quadtrees in which recalculated clusters are stored according to the zoom level. To reduce the size of the response message and therefore reduce the network transfer time, only required information like the cluster- or POI-ID and names are returned to the client. Detailed POI or cluster information will be handled in a own request if required.

Ontology-Search also uses the previously mentioned opening and core hours to show only currently opened places or on a time-picker selected date opened places on the map and in the hit list.

3. Route Planning

In this chapter, we first describe how to use Dijkstra to compute routes over POIs with time windows. Subsequently, we introduce more involved techniques to solve our problem which have the potential to be much faster in practice.

3.1. Basic Dijkstra

All shortest path calculations in this thesis are based on the Dijkstra algorithm. During a Dikstra run, the following information are stored: the distance from each vertex, of the graph to the source, the predecessor of each vertex and a min priority queue sorted by the distance. During the initialization, the distance of the source has to be set to zero while all other distances are set to infinity and their predecessors are undefined. The source is added to the queue. In the main loop, each time a vertex with the minimum distance is extracted from the queue, the distance to its neighbours will be calculated and updated if a shortcut was found. If the queue is emptied, the distance to each vertex in the graph connected to the source is calculated. The path can be extracted by following the predecessors up to the source.

Since the opening hours are important for the calculations later on, the travel time was used as metric determining the optimal path. This makes it possible to simply add the duration of stay or the waiting time before a node opens to the current costs. It also simplifies the open state testing of a node, since the current costs to a node also include the time to the previous nodes and the time the user would stay there.

The Dijkstra is a simple approach to calculate the shortest path from a source to all vertexes in a graph. To solve the problem of sequenced route planning, a naive approach will require multiple Dijkstra calculations. On a query, from source to target via a bakery and a butcher, a first Dijkstra is needed from the source to all bakeries. Then a Dijkstra from each bakery to all butchers and so on. This result in 1 + m + n Dijkstra runs where m being the number of bakeries in the graph and n the number of butchers. On the OSM-file of Germany, this would be $1 + 28.251 + 8.859 = 37.111$ Dijkstra calls. Two variations are required to avoid these multiple Dijkstra calculations.

Multi Target Dijkstra:
If a particular target is needed, the calculation can be aborted as soon as the target has been extracted from the priority queue. Handling multiple targets can be archived by adding them to a list and deleting a target as soon as it has been extracted. As soon as the list of targets is empty, all targets have been reached and the calculation can be aborted.

Multi Source Dijkstra:
To handle multiple sources, all start distances of the sources must be initialized with zero. The graph will then be explored from all sources at once and after the calculation the shortest path from one of the sources is revealed.

3.2. Sequenced Route Planning

The Multi Target/Source Dijkstra shown in the last section are key to calculating a route with different stopovers. This is done in different steps/layers. In the following, we will see these calculation on the basis of a request from source to target with a stopover at a bakery and a butcher. How a list of bakeries and butchers is extracted can be seen in the ensuing sections. Figure 5.5 illustrates this request in a layer view.

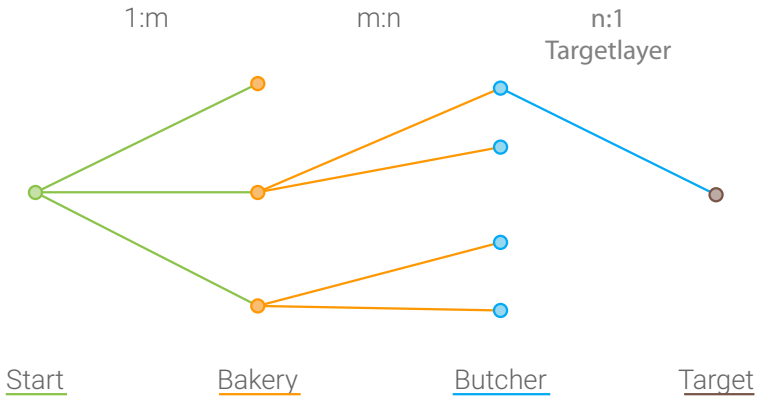


Figure 3.1.: layer view:

- The first layer is a 1:m Dijkstra where m is the number of bakeries within a specific radius around the start. Dijkstra will compute the distances to all bakeries within the radius. If a route with specific time parameters is desired, only nodes opened on the time of arrival and during the duration of stay will be handed over to the next layer. Alternatively, the time until the node is opened can be added to the distance.
- The second layer will be a m:n Dijkstra where n is the number of butchers within a specific radius around each of the previous calculated bakeries. The

Dijkstra will be initialised with the calculated bakeries from the first layer and their corresponding distances to the start and calculated from there on.

- The last layer uses a n:1 Dijkstra calculation from all previous calculated butchers to the destination. The calculation reuses the calculated butchers and their distances again. The shortest path is now observable.

To reduce memory consumption, only one Dijkstra-object is used. After every calculation the used variables will be set to their initial values.

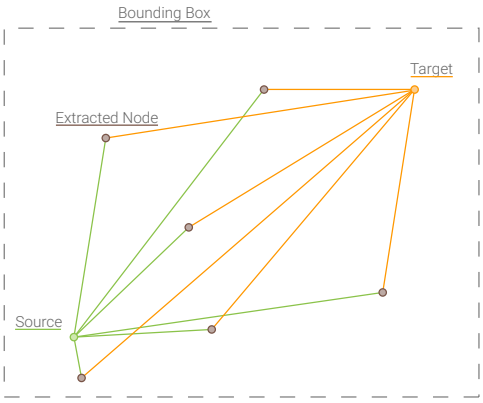
The variables m and n used in Figure 5.5 are already a heuristic improvement since the naive version uses all bakeries and butchers in the graph. To reduce the amount of POI used for the calculations, two approaches have been tested. The Bounding Box and the Iterative Doubling approach which will be described in the following sections.

3.3. Dijkstra Enhancement: Bounding Box

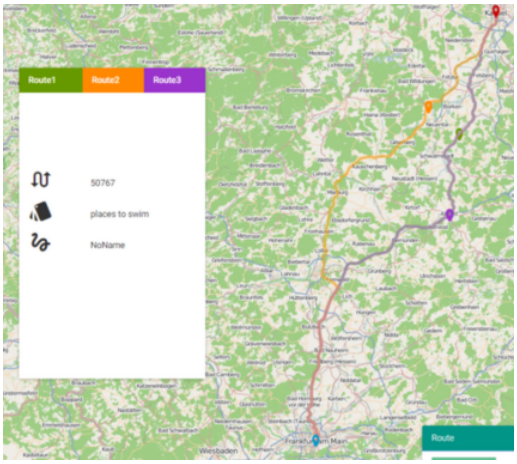
The bounding box approach was used as a first approach to calculate a route with stopovers. The approach uses the structure of the quad tree in each group and extracts the desired nodes within a bounding box. This bounding box was constructed around the source and target and extended about 20% of its size. The extracted POIs were used by a multiple target Dijkstra from the source to get the distance from source to each of the POIs. To get the distance from each POI to the target a multiple source Dijkstra from these extracted POI to the original target was calculated in an inverted graph. The smallest combination of the distance from source to a node (green lines in Figure 3.2a) and target to a node (red lines in Figure 3.2a) will be the calculated result. This approach also offers alternative routes if the second or third best combination is considered. But compared to the approach in the next chapter, the bounding box approach is a heuristic approach that does not have to result in the exact route.

If more than one stopover is desired or no alternative routes are required, the second layer can be initialized as mentioned before in Section 3.2 by the distance calculated to the nodes in the layer before. If no route has been found via the stopovers, the bounding box will be doubled and the calculation will be repeated. This behaviour is similar to the iterative doubling approach described in the following section.

Because of the extraction of POI with a bounding box, this heuristic approach could ignore the optimal path which may including a POI outside the bounding box.



(a) Figure showing the principle of the bounding box approach



(b) Screenshot of multiple route results

Figure 3.2.: Two figures of the Bounding Box Approach

3.4. Dijkstra Enhancement: Iterative Doubling

This section will introduce the used graph and the improvements that have been performed to optimize the extraction of nodes within the graph used in the Iterative Doubling.

Graph and NodeLinks

Vertex

The information stored about a vertex are its latitude and longitude position and optional NodeLinks (link to POI which have their entrance on the vertex). To easily calculate the nearest vertex of a point given its latitude and longitude, vertices are saved in a quad tree as well. On each query an entry point into the graph from the given position of source and target is extracted from the quad tree.

Edge

Each edge includes its distance in meter and the travel time and is saved in an offset array.

NodeLinks To prevent an extraction of the POIs within a bounding box each vertex could own one or several NodeLinks. These NodeLinks have two attributes:

- The **OSM-ID** of the POI we link to. The vertex which owns this NodeLink serves as an entry point into the graph for the POI.
- And a **GroupCode** to efficiently distinguish between a bakery and a butcher OSM-node. Each group of the ontology got a GroupCode assigned.

We can now check during the exploration each discovered vertex for linked OSM-nodes which are member of the requested group.

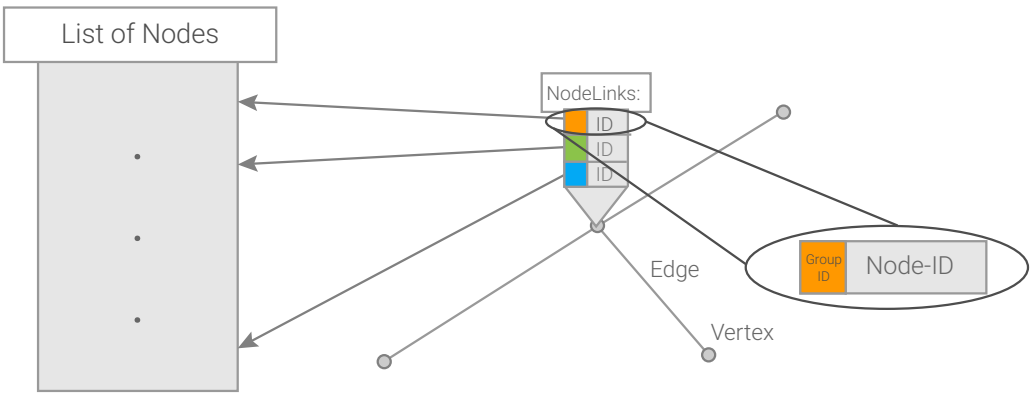


Figure 3.3.: NodeLinks

GroupCode

The GroupCode was implemented as a Char array. During the extraction of the Ontology, each group got a GroupCode assigned beginning with the root. This GroupCode enhanced each time it is handed down to its subgroups, which adds another char at the end. With this method, subgroups or parent/grandparents can be easily detected. This is required because all “Food”- nodes (GroupCode: [A]) include all ”Italian Restaurants”-nodes (GroupCode: [A,A,C]).

Using the NodeLinks with Dijkstra

The previously declared NodeLinks and GroupCodes are now used during the Dijkstra calculation. After the extraction of a vertex, the nodes linked with the vertex are checked if they are either members of the group or members of a subgroup currently searched for. This is done by checking its GroupCode. If the node is applicable, the current saved distance (in seconds/10) is used to check via the stored opening hours if the node is open upon arrival. Now the node is of the data opened or closed.
If the node is open, the duration of stay is added to its distance and tested if it’s also open during departure. The node is then saved as an ExploredNode containing the distance in seconds/10, the vertex at which the node is located and the source used to find the node.
If the node is closed, the remaining time until the shop opens will be added as waiting time. An ExploredNode will be created as mentioned before now containing an additional waiting time as well as the distance constructed by the distance to the node, the time until the node opens and the duration of stay. The ExploredNodes are used in the next layer as sources.

Iterative Doubling

The Iterative Doubling was introduced in the paper "Sequenced Route Queries" from Jochen Eisner and Stefan Funke [EF12]. They expect realistic sequenced route queries to be mostly local. Searching for facilities above 60 to 90 minutes distance to the source is rather inefficient. Their basic idea is to try to calculate a result within a smaller time frame and repeat it on failure within a greater radius. This will calculate optimal results in contrast to the bounding box approach.

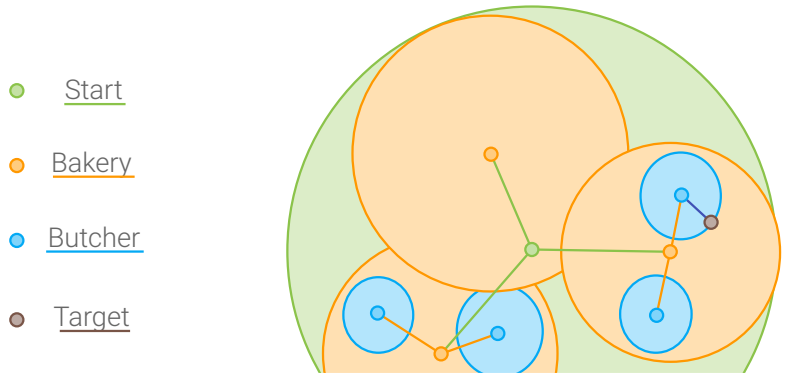


Figure 3.4.: layer view:

Figure 3.4 shows one run of a sequenced route planning. The route between the green start point and the brown target via a bakery (red dots) and a butcher (blue dots) should be found. In the first layer, the green circle around the start point indicates the used search space. The Dijkstra will be explored until the defined maximum distance is reached. After the first layer calculation, the found bakeries (red dots) will be used as start points for the second layer. The red circles around the bakeries indicate the used search space during this calculation. The radius of each red circle will vary since each start point will be initialized with the distance to the green start point. The radius of a bakery is the used maximum distance minus the distance from the green start point to the bakery. From these found butchers the rest of the maximum distance will be explored until the target is found or the radius is reached. If the radius is reached, the whole calculation is repeated with the doubled radius and so on.

Compared to the original Iterative Doubling, the implemented one using time windows will add the duration of stay of previous stopovers to the maximum distance of the current layer. The first layer would have the usual maximum distance, the second one would have the usual one plus the duration of stay of the first stopover. Given a usual maximum distance of 30 minutes and a duration of stay at the bakery of 10 minutes, the maximum distance of the first layer would be 30 minutes and 40 minutes for the second layer. However, since the duration of stay has been added to

the costs of all reached bakeries as well, the sources in the second will be initialized with a distance greater then the 10 minutes of the bakeries duration of stay and therefore is the search space of the second layer is still 30 minutes.

Iterative Doubling is optimal because it is based on the same principle as the original Dijkstra. After each new processed vertex, the distance to this vertex is guaranteed to be the shortest one. Iterative Doubling attempts to avoid unnecessary calculations after finding an optimal solution.

3.5. Iterative Doubling Enhancement: Nearest Group Member Bounds

In this section a dynamic initial radius for the iterative doubling will be introduced.

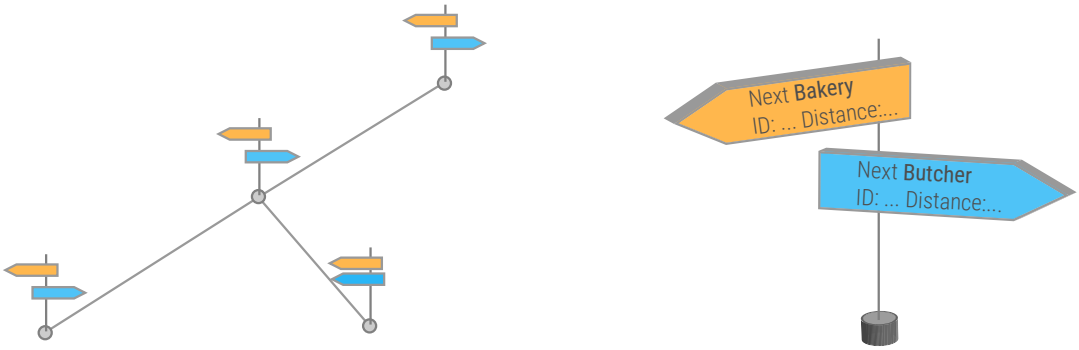


Figure 3.5.: layer view:

Queries in which a stopover is a very sparsely represented facility, such as a specific electronic consumer market or a special clothes company, will cause the iterative doubling to recalculate a couple of times to get a useful result because the used first maximum distance was too small. Imagine the start point to be in the middle of the Black Forest. The “MediaMarkt” nodes are located around the black forest which will result in more than an hour of driving to reach the next one. The initial maximum distance of say 30 minutes won’t result in a successful calculation twice (the initial one to 30 minutes and the one to one hour) . This means the radius of 30 minutes around the source was calculated at least three times and the radius of one hour was calculated twice to reach the “MediaMarkt”. Lets assume we would know the next “MediaMarkt”’ of the source point and it’s distance. We could then initialize the starting maximum distance of the Iterative Doubling with at least this distance and save the unneeded recalculations.

To do so, we pre calculate for each vertex the next member of each group and its distance. This can be done efficiently by using a reversed graph and adding the members of the group to be calculated to the source list. We have now attached a signpost for each group to the vertex showing the distance and the vertex-ID of the nearest member of that group as illustrated in Figure 3.5.

On query time, the maximum distance for the first Iterative Doubling Dijkstra will be calculated by using the saved distance to the “MediaMarkt” and a Dijkstra calculation from this “MediaMarkt” to the target. If multiple stopovers are acquired (for example “MediaMarkt” and afterwards “Orsay”) the next pre calculated “Orsay” from the used “MediaMarkt” vertex can be used to extend this bound.

A result is already found even by just using the preprocessed nearest group member, even if it isn’t the best one. Executing the Iterative Doubling with these upper bounds will return at least this result or even better without multiple Iterative Doubling recalculations. In the case of using the time window, this enhancement depends largely on the open state of the stopovers.

4. Server and Client

This chapter introduces the two components of the system, namely, a Java-written Server providing the necessary information to be shown and an HTML/CSS/JS based Client able to visualize the requested information.

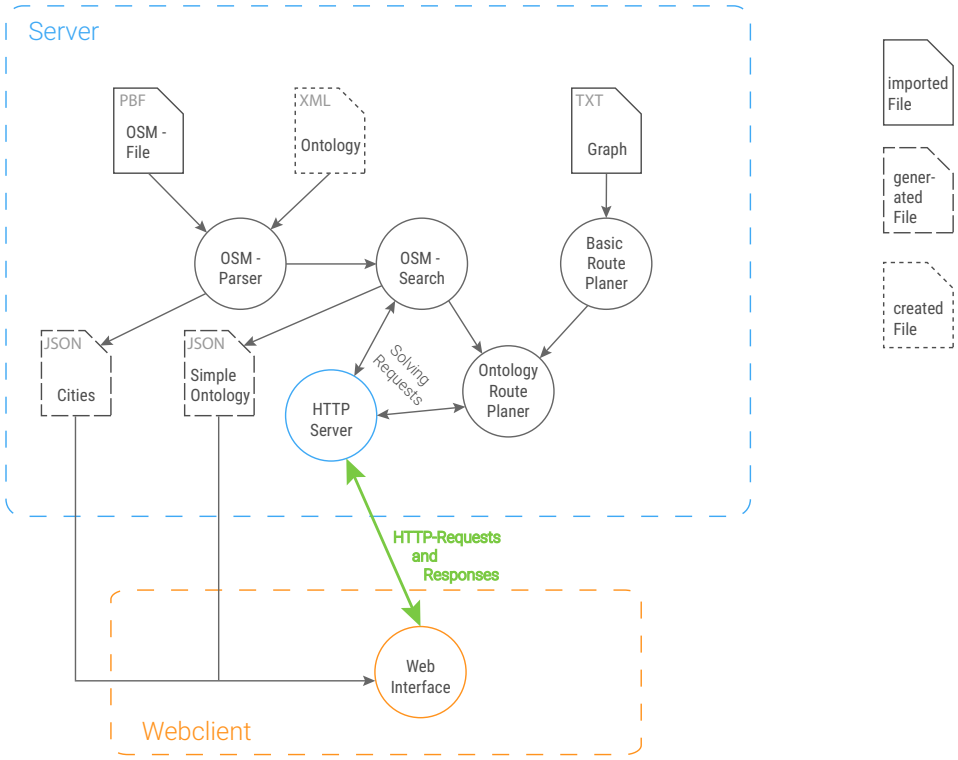


Figure 4.1.: Overview

Figure 4.1 shows the overview of the system. On server start the OSM-Parser loads the provided OSM-File and creates a cities.json file containing all bigger cities and their latitude/longitude position, which is required by the Web Interface to zoom in to a specific city. The OSM-Parser also parses nodes containing the tags defined within the ontology file. These nodes are saved in a quad tree data structure and clustered to offer an OSM-Search. The OSM-Search also generates a simple version of the Ontology File in JSON format to be loaded by the Web Interface.

A graph will be constructed from the Graph file to build a Basic Route Planer. The Nodes and their positions stored by the OSM-Search will be combined with the constructed Basic Route Planer to build an ontology-based Route Planer.
Once all files are loaded and preprocessed, the HTTP Server will start and distribute the requests to the OSM-Search and Ontology Route Planer.
The Web Interface loads the list of cities to offer a search for cities. It also loads the Simple Ontology to show the available groups.

4.1. Server

Aside from the data-request used to get all clusters within an area the node-request used to get detailed information about a node given its node-ID and the statistics-request, used to get the information displayed on the statistics.html page, the path-request is the one used to handle the route planning. The HTTP-interface requires the following parameters:

- the latitude and longitude of the source and target as lat1=[source latitude], lat2=[target latitude]
- the starttime if used, given in the dd.MM.yyyy hh.mm - format
- the stopovers as via1=[groupname] via1time=[duration in minutes]

After calculating the best route and stopovers, the server will return a response in JSON providing the required information:

- the request parameters
- the calculated route giving:
  - the subpaths (latitude and longitude of each graph-node) and their distance and travel time
  - start time and time of arrival
  - detailed information about the stopover nodes including name, website, phone number, etc.
- occurred errors or warnings

If an error takes place during the calculation, an error message will be added to the server response and pop up via a snackbar toast within the web client to inform the user. For example, in case of used core hours to determine the open state on arrival, the user should be informed that no guarantee can be given on the actual open state.

4.2. WebView

After introducing the server side in the last section, this section will handle the used frameworks on the client side.

The web client builds on the framework Bootstrap introduced by Twitter. Bootstrap is an HTML, CSS, and JS - framework which provides custom HTML and CSS components and jQuery plugins like Typeahead[1], used for auto-completion search of cities and ontology-groups, Datetime Picker[2], a plugin for picking the start time, and SnackbarJS, which creates Material Design like snackbars and toast. A material design theme[3] based on the Google Material Design from Google[4] was used to boost the simple look.

The following screenshot shows the main use of the user interface. On the top corner, a navigation bar can be found giving the possibility of changing between the route view and the previously mentioned Ontology Search. The typeahead plugin in the middle of the navigation bar offers an autocomplete search for cities to zoom to. On the ride side of the navigation bar is the tile selection, offering different map tiles like the Mapnik or Mapquest tiles.

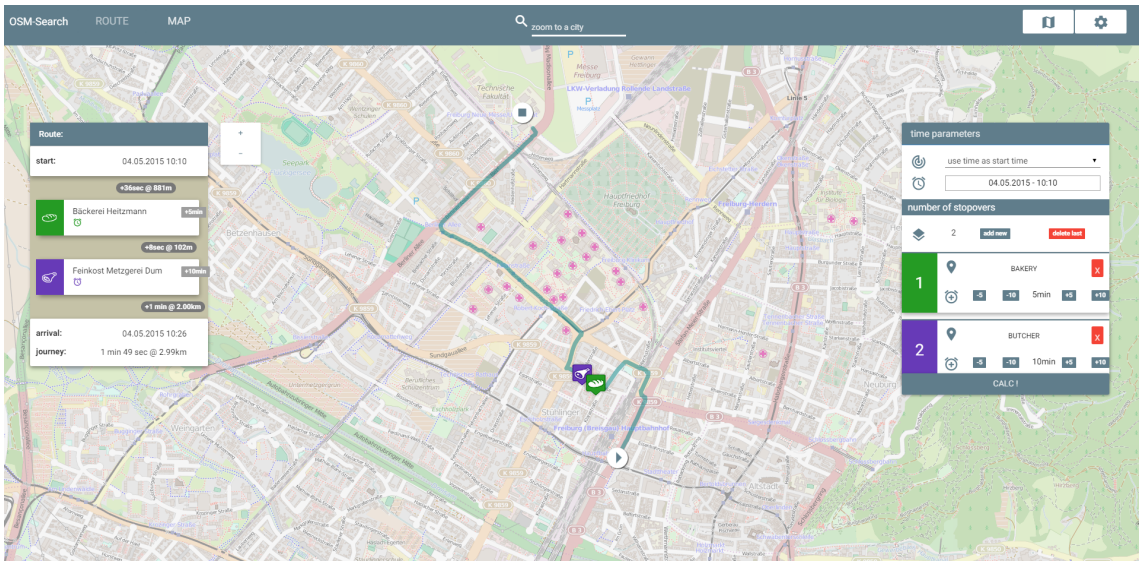


Figure 4.2.: route overview

[1] https://github.com/bassjobsen/Bootstrap-3-Typeahead
[2] http://www.malot.fr/bootstrap-datetimepicker/
[3] http://fezvrasta.github.io/bootstrap-material-design/
[4] http://google.com/design/spec/material-design/

The time parameters and stopovers can be found on the ride side of the overview, which contains a dropdownbox to give the option of either using the selected time as start time for the calculation, or ignoring open hours and just calculating the fastest path. The DateTimePicker plugin offers a nice and intuitive way of selecting the start time by simply picking the day, the month and hour separately. In the stopover selection, up to four stopovers can be chosen along with the requested group and the duration of stay. Each stopover will receive a color which can be found on each reference of this stopover like the marker on the map and in the route detail view.

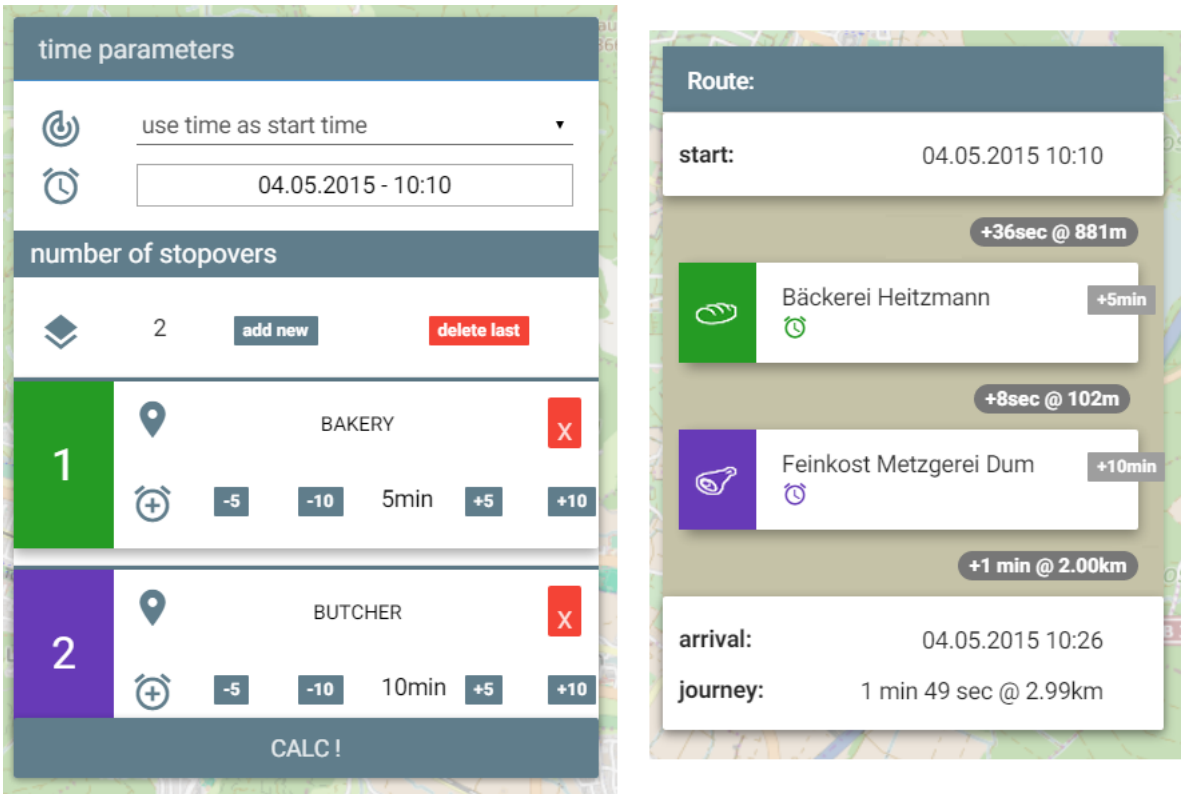


Figure 4.3.: left: stop overs, right: route details

The route details view, found on the left side of the overview, presents detailed information about the calculated route, like the start date and the time of arrival of the whole trip as well as of each stopover. The distance of the journey and the distance between each stopover in meters is also shown. Small boxes offer short information about each approached POI, which can be extended via a toolbox triggered on mouse over to get the exact opening hours for example.

Another big component of the webclient is the mapping library Openlayers[5], providing a fast and easy to use interactive map and data visualisation layers. Openlayers is compatible with many different tile servers like the Google Maps, Yahoo Maps or Bing Maps tile layers. Other different layers containing the source/target and POI icons or the route can be visualized as well. Each destination is highlighted with the coloured marker deposited in the ontology. These markers show an icon that can be associated with the corresponding group (The bakery icon shows bread,...).
To change the source or target of a route, the start and stop markers can be moved via drag and drop or just by right clicking somewhere on the map and selecting the position as start or stop.
To simplify the selection of frequently used start or stop address, the position of the home and work address can be saved as cookies. From then on, just a right click and a selection is needed to set the start or stop to the desired position.
The parameter in the url of the page will be synchronised with the performed actions, which enables easy sharing of calculated routes with others. A url could even be generated out of another application, including the current position for example.

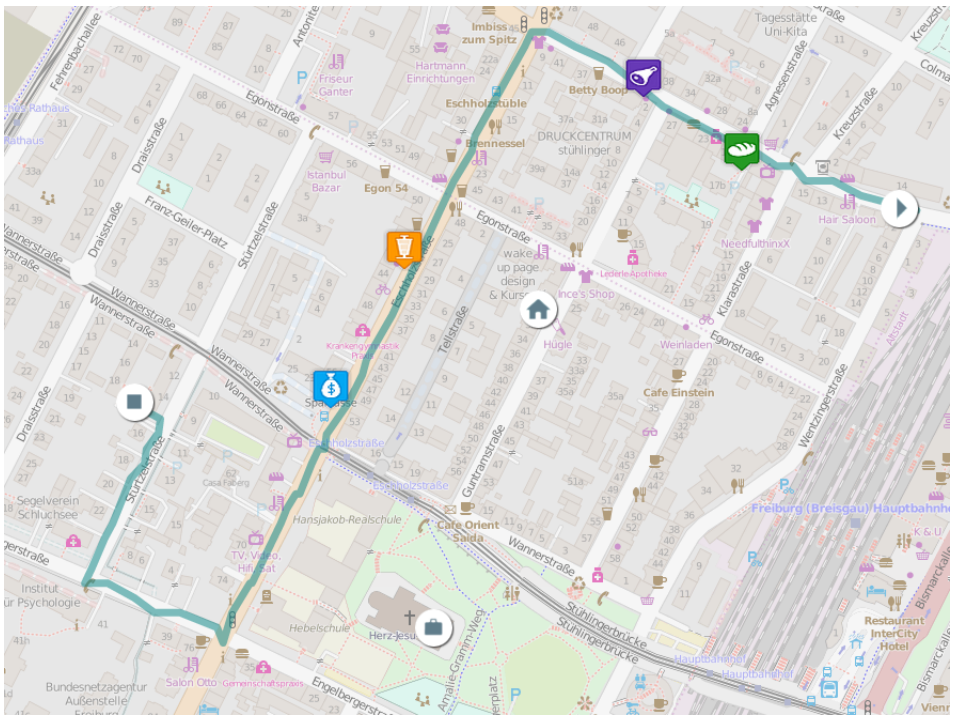


Figure 4.4.: Route via a “Bakery”, “Butcher”, “FastFood” and “Bank” POI

[5]http://openlayers.org/

5. Experiments

In this chapter experimental tests are described and the results are presented.

5.1. Test Cases

To validate the improvements during the development, different test sets have been created. Two test areas have been used. One is a local test area located in a radius of 25 km around Stuttgart to test small queries as well as high node density areas and the other is a bigger test area containing whole Baden Württemberg. Within these areas, random source and target positions are created for each test execution during runtime. These source and target positions have no minimum distance to each other, but rather a maximum distance created by the dataset.

Different test sets with different number of stopovers assure a wide test spectrum. To test queries with the same source and target, a test called "Sunday morning bakery" was prepared, which involves just running to the next bakery on Sunday morning and back. The "Weekly Shopping Tour" got a source=target test as well. These test sets have been saved in a JSON file containing all required parameters and results.

The following seven test cases have been created:

- **Sunday Morning Bakery:** A test queue performed on a Sunday morning at 9 leading to the next bakery and back to the origin.
- **After work gas station:** A single stopover queue performed at 6pm as a local and global route from work back home via a gas station.
- **Florist and Hairdresser:** A local and global test set featuring two stopovers with a low density.
- **Bakery and Butcher:** A local and global test set featuring two stopovers with a high density.
- **Bank, Clothes and Discounter:** A local and global test set featuring three stopovers with a high density.
- **ALDI, LIDL and Burger:** A local and global test set featuring three stopovers with a lower density.

- **Weekly Shopping Tour:** A local and global test set featuring four stopovers (Bio Market, REWE, DM and Media Markt). Since a weekly shopping tour would be performed from home, this test set will also be executed with the same source and target.

The server executes the tests n times and creates a performance.json file containing the results of each run as well as the average of each test. Included in the results are:

- the runtime
- the random start and stop and its distance (for reconstruction)
- runtime and number of settled nodes of each layer

performance.html

These performance files can be loaded into an HTML file visualizing the results via charts provided by the CanvasJS framework [1]. The two screenshots below show the overview chart, which contains the average runtime during the development, and a detailed view of one test.

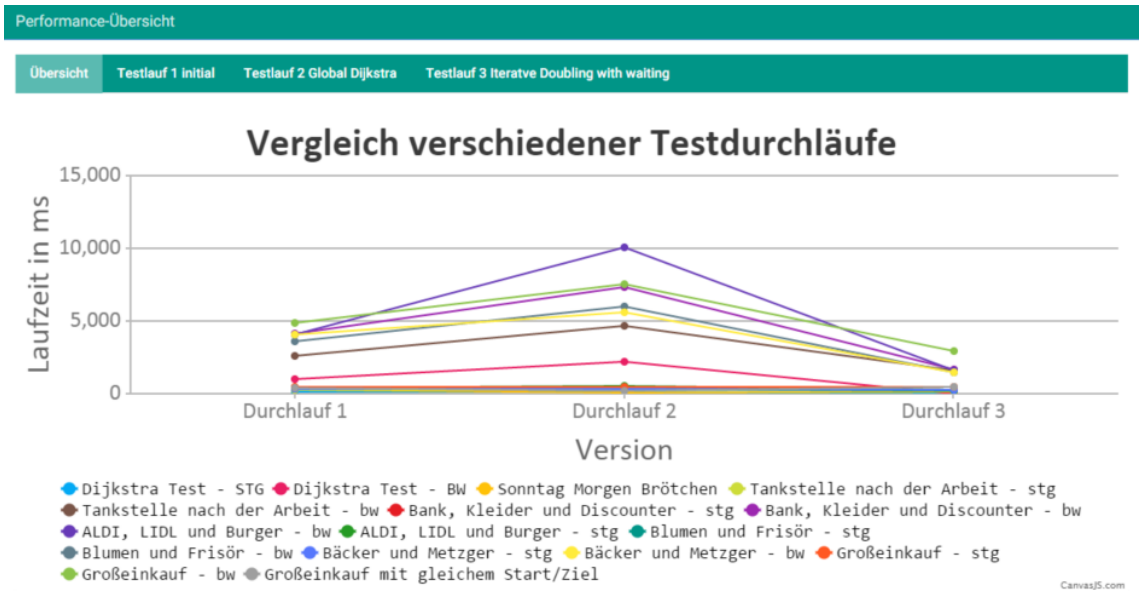


Figure 5.1.: performance overview

[1] http://canvasjs.com

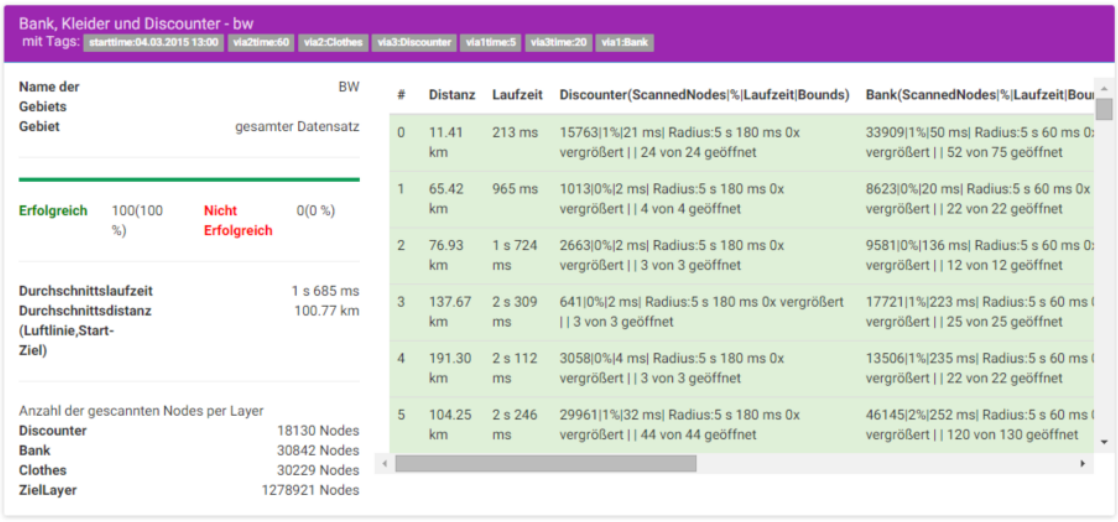


Figure 5.2.: performance details

5.2. Evaluation

The previously declared tests have been executed 100 times. The running times are the times needed to calculate the required stopovers and the total journey time. To get the complete path n+1 (while n is the number of stopovers) 1-to-1 Dijkstras need to be performed. Because of the memory consumption on a desktop pc the tests have been performed with the Baden Württemberg dataset. This dataset consists of 2.459.354 vertices, 4.993.582 Edges and 80.458 OSM-Nodes and was executed on one core of an i7-3630QM CPU.

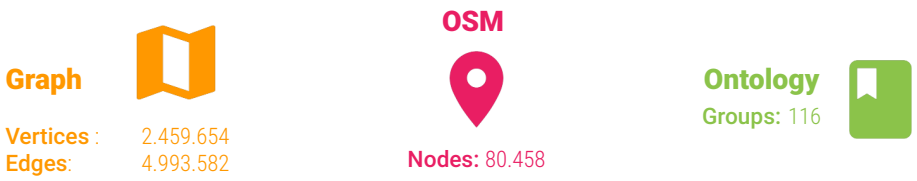


Figure 5.3.: Statistics

The following performance times have been archived:

Local Test - Stuttgart Area

Average straight line distance from source to target: 12 km.

Name	Stopovers	BoundingBox	Iterative Doubling
Sunday Morning Bakery*	1	251 ms	113 ms
After Work Gas Station	1	234 ms	180 ms
Florist and Hairdresser	2	340 ms	217 ms
Bakery and Butcher	2	361 ms	256 ms
Bank, Clothes and Discounter	3	384 ms	279 ms
ALDI, LIDL and Burger	3	393 ms	266 ms
Weekly Shopping Tour	4	458 ms	473 ms
Weekly Shopping Tour *	4	429 ms	482 ms

Figure 5.4.: Statistics

Tests marked with an * have the same source and target.

Baden Württemberg Area

Average straight line distance from source to target: 100 km.

Name	Stopovers	BoundingBox	Iterative Doubling
After Work Gas Station	1	2 s 631 ms	1 s 633 ms
Florist and Hairdresser	2	3 s 615 ms	1 s 462 ms
Bakery and Butcher	2	4 s 64 ms	1 s 441 ms
Bank, Clothes and Discounter	3	4 s 126 ms	1 s 685 ms
ALDI, LIDL and Burger	3	4 s 71 ms	1 s 606 ms
Weekly Shopping Tour	4	4 s 848 ms	2 s 959 ms

Figure 5.5.: Statistics

A difference between the two areas is not just the straight line distance from source to target, but also the density of available POI. While the area of the local test is full of possible POI, since it consists of a big city, the bigger test has to either collect the POI on the way or if source and target are located inside of bigger city areas the POI in a big city are visited along the way.

The improvement on local queries is enough to ensure interactive route planning. Iterative Doubling not only improve the performance, but also allows optimal results

compared to the bounding box approach. In the "Weekly Shopping Tour" test case of the local test, the bounding box approach seems slightly better. The difference is not really significant, but it could be solved with parameter tweaking. The bounding box approach uses a dynamic initial radius, whereas the iterative doubling has a static initial radius (30 minutes). In Section 3.5 a dynamic initial radius for iterative doubling was introduced.

Influence of the starting time

The used test cases were initialised with more or less optimal starting times on execution. For example, the "Sunday Morning Bakery"-test was executed with the starting time of 9 am, whereas the same test executed at 9 pm would result in a much worse running time because no bakery will be opened after 9 pm on Sunday in Stuttgart. To show this effect, the "Florist and Hairdresser" test case has been run within the Stuttgart area for each full hour from midnight to 11 pm to test the running time changes during a day. The used core hours of the group Hairdresser are "Mo-Fr 09:00-18:00" whereas for the Florist group they are "Mo-Fr 09:00-12:00, 14:00-18:00". The line chart Figure 5.6 shows the changes of the running time depending on the used start time using Iterative Doubling. Between 7 am and 7 pm a quick solution can be found. During the core hours of both groups ("Mo-Fr 09:00-12:00, 14:00-18:00") every POI of both groups is available, which results in fast calculable solutions. Between 5 am and 7 am and between 7 pm and 9 pm, solutions get rare and a lot of driving is necessary which results in a longer run time. During the night the calculations search the entire graph, which explains the long running time.

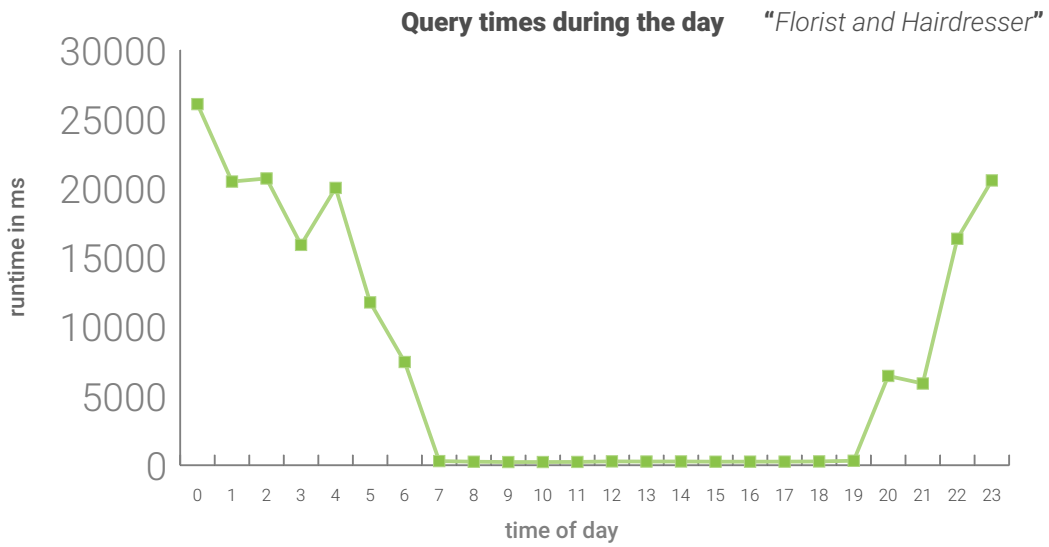


Figure 5.6.: Running time during the day for the "Florist and Hairdresser" query. The running time between 7 am and 7 pm is at around 250 ms.

6. Conclusion and Outlook

A route to our birthday barbecue party on time can be found in time. While a naive approach would take more then 20 seconds [EF12] the Iterative Doubling versions returns an optimal result within less than half a second for local queries. This improvement could be increased by using acceleration technique for route planning as contraction hierarchies. Which should result in a speedup of factor 15 according to Eisner and Funke [EF12].

The results vary with respect to the starting time because it affects the time windows. The running time increases on the late evening, night and early morning because almost every shop is closed. During the core hours the best running time (and probably shortest route) can be achieved since most shops are opened. This supports the influence of the opening hours and confirms the importance of the core hours. But if the used core hours are not precise because the opening hours of the POI group too much, the whole calculation will be incomplete and no guarantee on the solution can be given. This is why the client warns the user on routes involving core hours.

Even if the experiments were executed on the Baden Württemberg data set the system works with the data set of whole Germany as well. The graph representing the road network of Germany consists of 21.721.465 vertices and 44.108.723 edges. The created ontology uses 549.429 POIs extracted from OpenStreetMap which are sorted into 116 groups.

The following possible implementation extensions that came up during the implementation:

- Adding public transport routing to search within train and bus connections or just walking or biking network graphs.
- Using car parks as routing points. They contain all the POI within a walking distance so going to a bakery and a butcher could be done with a single parking. Multi criteria search for not only the shortest path but also the one least likely to encounter trouble searching for parking could be useful. The real-time status of the car park with the number of unused parking lots could be integrated.
- A mobile web page that offers predefined queries like “get me to the next open bakery and back” which could easily be build given the current position. Provided that the home or work address is deposited as a cookie, even queries like "’get me home from here via a supermarket that will be open on arrival"’ are done with a single click.

The created ontology already handles over 100 groups, but a lot of different POI types like tourist attractions are not integrated yet because they are typically not in these kind of routing queries. The Open Street Map project is evolving and new POI types will be added and old ones will be updated, in which case the created ontology could be extended and enhanced.

Because of the lack of opening hours saved in Open Street Map an additional source of opening hours or an improvement of the OSM-data is needed.

A. Ontology Map

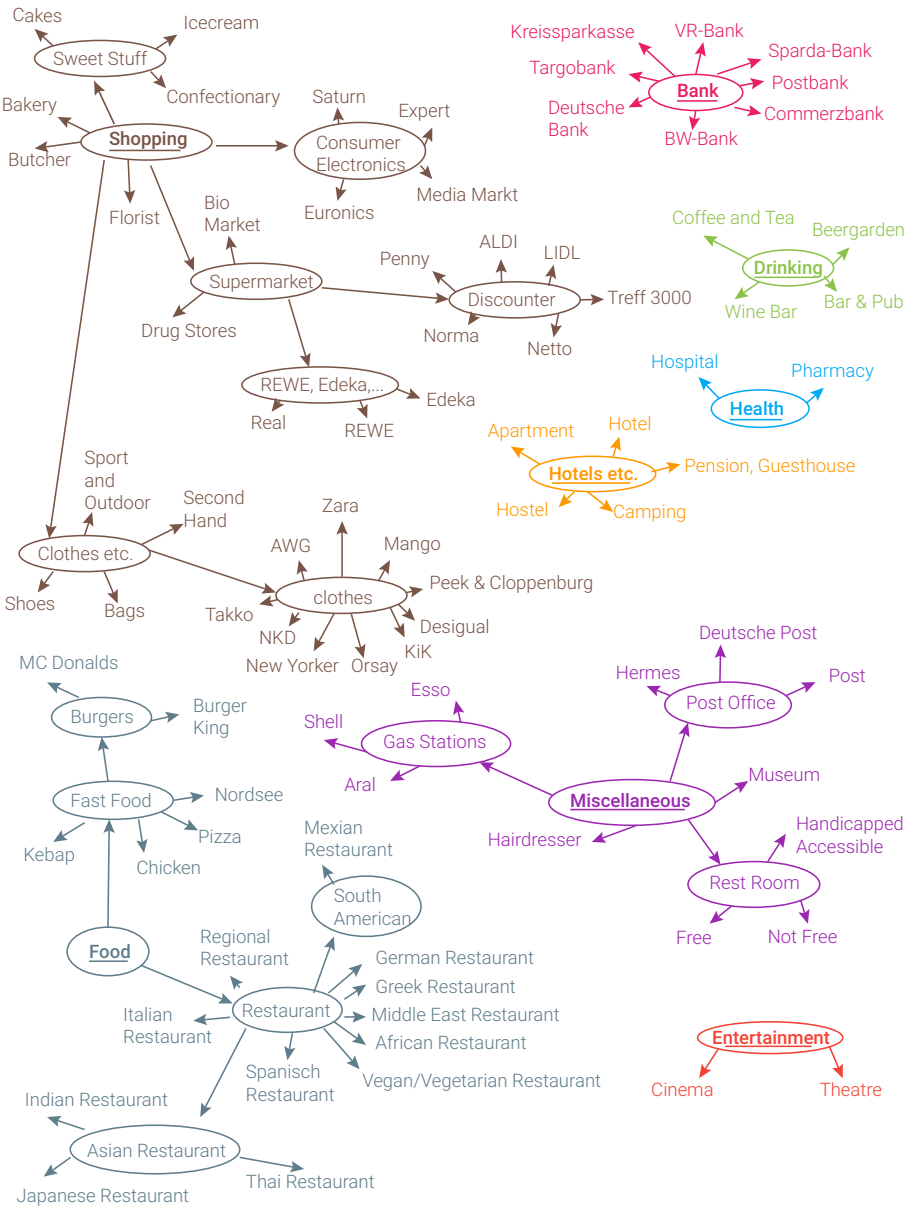


Figure A.1.: Ontology Map

Bibliography

[CHK+11] Codescu, Mihai ; Horsinka, Gregor ; Kutz, Oliver ; Mossakowski, Till ; Rau, Rafaela: DO-ROAM: Activity-Oriented Search and Navigation with OpenStreetMap. In: *GeoSpatial Semantics - 4th International Conference, GeoS 2011, Brest, France, May 12-13, 2011. Proceedings*, 2011, S. 88–107

[DGLT12] Dash, Sanjeeb ; GÃŒnlÃŒk, Oktay ; Lodi, Andrea ; Tramontani, Andrea: A Time Bucket Formulation for the Traveling Salesman Problem with Time Windows. In: *INFORMS Journal on Computing* 24 (2012), Nr. 1, S. 132–147

[EF12] Eisner, Jochen ; Funke, Stefan: Sequenced route queries: getting things done on the way back home. In: *SIGSPATIAL 2012 International Conference on Advances in Geographic Information Systems (formerly known as GIS), SIGSPATIAL'12, Redondo Beach, CA, USA, November 7-9, 2012*, 2012, S. 502–505

[GSSD08] Geisberger, Robert ; Sanders, Peter ; Schultes, Dominik ; Delling, Daniel: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In: *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, 2008, S. 319–333

[MCM12] Mihai Codescu, Oliver K. ; Mossakowski, Till: Ontology-based route planning for OpenStreetMap. 901 (2012), S. 62–73

[Ope15a] OpenStreetMap.org. *OpenStreetMap.org*. 2015

[Ope15b] OpenStreetMap.org. *Wiki.OpenStreetMap.org*. 2015