Algorithms for OpenStreetMap Data
- Wikipedia Round Trip -

Florian Strohm
University of Stuttgart
florian.strohm@stud.uni-stuttgart.de

May 3, 2018

Figure 1: Example round trip visiting the selected Wikipedia articles (green markers).

1 Introduction

When visiting a city for the first time, we often want to explore the city and visit points of interest (POI). But to decide which places to visit and especially in which order can be tedious because we first have to search for different POIs, look up their location, memorize all of them and find a practical route to visit them all. The other option is to buy guidebooks, but they cost money and most of the time only consider the most popular points of interest. Also, you still have to navigate through the city by yourself.

In this work we present a sightseeing round trip planer tool based on Wikipedia[1] articles and OpenStreetMap [HW08] graph data, which helps to easily find POIs. Wherever we are, with this application we are able to find POIs nearby and calculate an optimal round trip on the fly. As can be seen in Figure 1, the user selected multiple Wikipedia articles and calculated a round trip. The shortest path visiting each article and going back to the starting point, called traveling salesperson problem (TSP), is visualized with a red polyline. One challenge of this project is to implement a performant algorithm for solving the TSP. Information about a specific article is shown on the left side. The article image and its text are directly retrieved from Wikipedia.

[1] https://www.wikipedia.de/

2 Architecture

To make the application easily accessible for many different users, we split it into a front-end for user friendly interactions and a back-end running on a server for heavy calculations. Figure 2 shows this two-tier structure and how the different components interact with each other. For easy full-stack application development we use the Spring framework[2]. At first we will discuss the role of the front-end in Section 3 and afterward the functions of the back-end in Section 4.

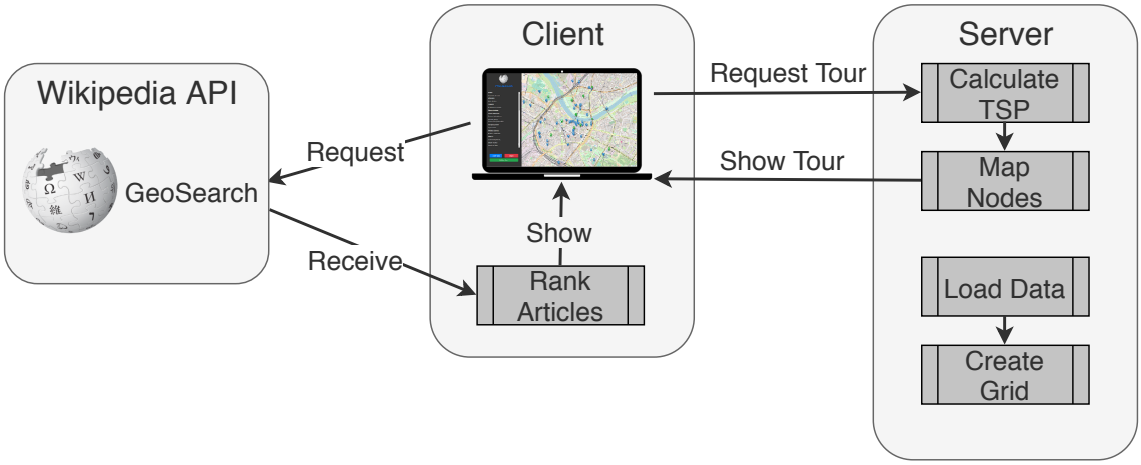

Figure 2: Architecture of the application.

3 Front-End (Client)

The front-end is realized in form of a web application to ensure easy and mobile access for multiple concurrent users. It is programmed using native HTML, XML and JavaScript.

3.1 User Interface

Figure 3 shows the start screen of the front-end. After a user opened the website, the map tiles are loaded and visualized by the Leaflet library[3]. If the website is allowed to read the users location, it zooms to the specific area. The sidebar on the left shows some instructions on how to use the application. The map shows a blue marker which indicates the point where to search nearby Wikipedia articles. The user can change the position of the marker by dragging or clicking the map.

After the user decided were to set the marker and clicked on the 'Get Articles' button, the front-end requests Wikipedia articles around the selected location from the MediaWiki API (see Section 3.2). After the front-end received all Wikipedia articles, it calculates a ranking and sorts the articles (see Section 3.2). Each Wikipedia article comes with geographic coordinates. As can be seen in Figure 4, a marker is created for each retrieved article using these coordinates. Only the top 30 articles are shown on the map. The user can click the button 'More Wikis' to show the next 30 articles (if possible). The sidebar on the left shows a ranked list containing the titles of the displayed articles. If the user left-clicks on a marker or on a title in the sidebar, the sidebar shows an extract of the corresponding article and its marker is colored red (Figure 1).

The user is now supposed to explore the different Wikipedia articles and decide which to visit. The user can right-click on the desired articles to select them and change the marker color to green. After the user selected at least two markers, he is able to calculate a round trip by clicking the button 'Calculate Tour'. By clicking this button, the front-end sends a request to the back-end containing the coordinates of each selected article via a REST API. The back-end calculates the tour (Section 4) and returns the result back to the front-end. The font-end retrieves the tour and

[2]https://www.spring.io/
[3]http://www.leafletjs.com/

Figure 3: Start screen of the web application.

visualizes it using a red polyline, as shown in Figure 1. The user can select more or delete markers and recalculate the tour, or start all over again by clicking the 'Reset' button.

3.2 Wikipedia Articles

Wikipedia is created using the MediaWiki[4] engine. The MediaWiki engine comes with a extensive REST API[5] allowing us to receive all information about articles we want. Whenever a user clicks the button 'Get Articles', the font-end sends a request to the MediaWiki API of the following form:

```
"https://de.wikipedia.org/w/api.php?action=query & list=geosearch &
gscoord=48.74559|9.10563 & gslimit=500 & gsradius=10000 & format=json"
```

The request says that we want to address the German Wikipedia (de.wikipedia.org) and that we want to query information (action=query). We further specify that we want to do a geographic search (list=geosearch), meaning to search articles around a specific point. This point is specified with a latitude|longitude pair (gscoord=48.74559|9.10563). We set these coordinates equal to those of the marker placed by the user. The attribute gslimit specifies the maximum retrieved articles and the attribute gsradius the maximum search radius. Unfortunately, the WikiMedia API limits us to an article limit of 500 and a search radius limit of 10000 meters. The returned articles from the API are stored in a JSON format as specified by our query (format=json). The problem is that the geosearch lists only ID, title and geographic coordinates for each article, but we also want its content and other metadata. Therefore we us the IDs of the retrieved articles to start another query of the form:

```
"https://de.wikipedia.org/w/api.php?action=query &
pageids=96114|1089300 & prop=extracts|pageviews|pageimages &
exintro= & explaintext= & pithumbsize=300 & format=json"
```

Again, we want to address the German Wikipedia and query information. This time we specify exactly which articles we want using the IDs (pageids=96114|1089300). We want to retrieve the text, images and the daily page views for each of the articles (prop=extracts|pageviews|pageimages). The attribute 'exintro' indicates that we only want the content before the first section, 'explaintext' says that we want the extracts as plain text instead of limited HTML and 'pithumbsize' specifies the resolution of the thumbnail images.

[4]https://www.mediawiki.org
[5]https://www.mediawiki.org/wiki/API:Query

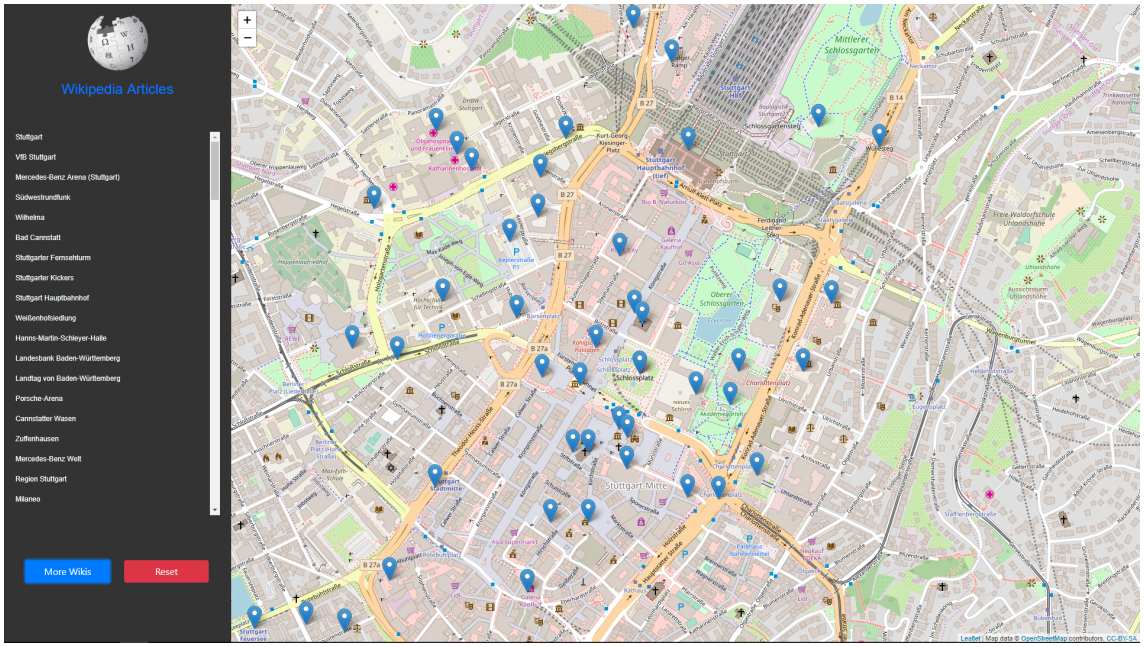

Figure 4: The sidebar shows a list containing found Wikipedia articles and the map shows a marker for each of these article.

The problem with this query is that the API limits us to 20 IDs (articles) per query. Therefore we have to do up to 25 requests if we receive the maximum amount of 500 articles from the initial query. If we start 25 synchronous requests, the front-end has to wait a long time, wherefore we work with asynchronous requests. After the front-end received the response from all requests it ranks the articles according to their average daily page views (remember that we requested this property in the query). After the articles are ranked, the front-end shows the top 30 articles as in Figure 4.

4 Back-End (Server)

The reason for the separation in a decentralized front-end and a central back-end is due to high performance needs for the round trip calculation. Another reason is that the large graph file has only to be loaded once on the server instead of multiple times on each client machine.

After the server is started, the back-end loads the graph data (Section 4.1) and creates a grid containing each node of the graph (Section 4.1.1). If it receives a request from the front-end via our REST API, it maps each latitude-longitude pair from the request to an actual node of the graph (Section 4.1.2). Finally the back-end calculates the shortest round trip visiting all these nodes (Section 4.2) and returns the result to the front-end.

4.1 Graph

The graph is extracted from the official OSM .pbf files using the Java library osm4j[6] as a preprocessing step. The round trip planer is designed for bike tours, wherefore we extracted all streets accessible with a bike. Due to main memory limitations we restrict the graph to contain only nodes and edges in Germany. Our complete parsed graph of Germany contains 124.989.086 edges and 57.715.697 nodes, using about 3GB of main memory when loaded.

4.1.1 Grid

After the graph data is loaded into the main memory, the back-end creates a grid using the nodes of the graph. Figure 5 shows an example grid containing several nodes. The location and size

[6] http://www.jaryard.com/projects/osm4j/index.html

of a grid cell are determined by a latitude and longitude tuple (lat|lon). We always round to the first digit after a comma. Each cell is 0.1 degree latitude wide and 0.1 degree longitude high, for example, there exist a cell starting at (48.7|9.1) and ending at (48.8|9.2). This yields a total of 4993 cells for our particular graph. Each node of the graph is now mapped into one of the grid cells using its latitude and longitude. For example, the node (48.74559|9.10563) is mapped to the grid stating at (48.7|9.1). In our case, the grid uses an additional 4GB of main memory.

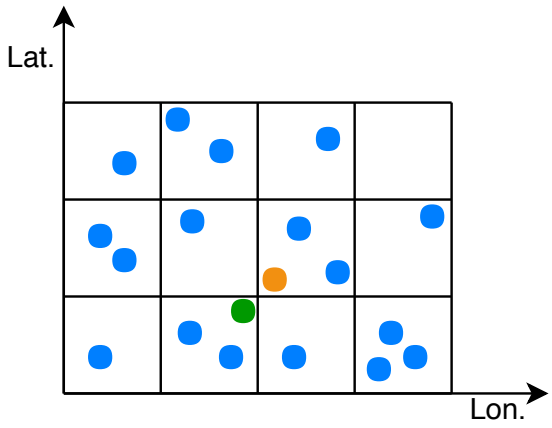

Figure 5: Example grid with geographic coordinates on the axes. The blue and green dots represent nodes of the graph, while the orange dot is representing some geographic coordinates, that shall be mapped to the nearest node in the graph (the green dot).

4.1.2 Node Mapping

In order to calculate a round trip visiting all user selected articles, we have to know which nodes in our graph to visit. Therefore, we somehow have to map the geographic coordinates of each selected Wikipedia article onto a node in our graph. To do so, we could calculate the distance of an article to each node of our graph and map the article to the closest node. The first occurring problem is that the earth is an ellipsoid, meaning we have to use the great-circle distance for distance calculation and also have to take care of irregular altitudes. Since this is computationally very expensive and we do not necessarily need a 100% accurate distance measure, we approximate the distance using the easy to compute euclidean distance. For short distances on earth the error of this approach is negligible and we are only interested in the relation between different measurements and not their exact value.

The second problem is that, as stated in Section 4.1, our graph consists of about 57 million nodes, meaning we would have to calculate 57 million times the euclidean distance to each node for each article. This would take far too much computation power, wherefore we use the previously created grid. If the back-end has to find the closet node of an article, it uses its (lat|lon) tuple to find the corresponding gird cell. Now it only has to calculate the distance to each node of this particular cell.

However, we have to take care of a special case shown in Figure 5. The orange dot represents the to be mapped article and the blue/green dots are nodes in our graph. We can see that the green dot is closest to the orange dot, while being part of a different grid cell. This means that the back-end also has to take into consideration all eight surrounding cells. Theoretically it would have to search even more cells if there is not any node within the first nine cells, but this does not happen in our case since the cells are fairly large. We can conclude the following formula for the average needed distance calculations Calc_{avg} :

$$\text{Calc}_{\text{avg}} = \frac{V}{C} * 9$$

with V being the amount of nodes in the graph and C the amount of cells in the grid. This reduces the average amount of calculations per article from about 57 million to $\frac{57}{4993} * 9 \approx 0.1$ million in our case.

4.2 Traveling Sales Person

The traveling salesperson problem (TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"[7]. We also call such a route a (shortest) Hamiltonian cycle.

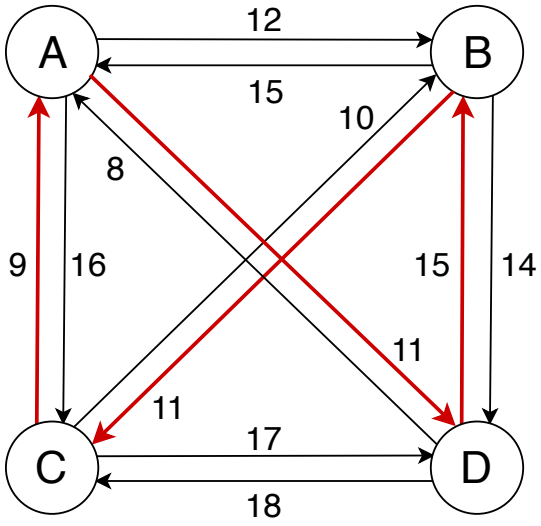

Figure 6: A complete, weighted, directed graph. The red arrows indicate the path of the shortest tour: A-D-B-C-A

More formally: given a complete, directed graph $G = (V, E)$ with a cost function $c : E \mapsto \mathbb{N}$, find a Hamiltonian cycle $H \subseteq E$ in G with $\sum_{e \in H} c(e)$ minimal. Figure 6 shows an example graph with $V = \{A, B, C, D\}$ and its Hamiltonian cycle $H = \{(A, D), (D, B), (B, C), (C, A)\}$ of minimum cost marked red.

The traveling salesperson problem is NP-hard, which means that under the assumption that $P \neq NP$ there cannot exist an algorithm that solves TSP in polynomial time. The naive approach to solve this problem is to calculate every possible tour and compare the costs. This brute force approach has a runtime of $O(n!)$ and therefore is not feasible in reality. We therefore use the more sophisticated approach of dynamic programming [Bel57]. As described by Held and Karp [HK62], and also by Bellman [Bel62], we split the TSP into multiple smaller subproblems. We solve the smaller problems and use them to calculate the solution for the original problem. This approach has a time complexity of $O(n^2 2^n)$, which is much better than $O(n!)$ but still very expensive. Solving the TSP with up to 25 nodes is possible within an acceptable amount of time, which appears to be enough for our use case.

If the back-end receives a request to calculate a round trip, it first calculates the distances (beeline) from each to each other received Wikipedia article. This distances represent the edge costs and are the input to the TSP solver. Algorithm 1 shows our dynamic, recursive program solving the TSP. Without loss of generality (since we calculate a round trip) we set the first selected article as our starting point. We use tables to keep track of the costs and picked nodes for each subproblem. For n nodes this tables are of size $n \times 2^n$, needing a lot of main memory for high n.

The first conditional clause of the algorithm is used to break out of recursion. We know that the last step is to go from any node back to the starting node and therefore already know the costs of the smallest subproblems on the bottom of the recursion tree. Then we iterate over each node of the node set and recursively calculate the costs of the subpath. After we iterated over every node, we know which one is the best to pick. We store the cost and the best node for each subproblem. After the algorithm terminated we can reconstruct the shortest round trip. Now we map each Wikipedia article in the round trip on a node in our graph as described in Section 4.1.1. Finally, we calculate the shortest paths from node to node in the order the TSP solver has calculated using Dijkstras algorithm [Dij59] and return the path back to the front-end where it is visualized for the client.

[7] https://en.wikipedia.org/wiki/Travelling_salesman_problem

Algorithm 1 Traveling Sales Person - Dynamic Program

```
1: procedure RECURSIVETSP(startNode, nodeSet)
2:     if subpath already calculated then
3:         return calculated cost
4:     end if
5:     cost ← maxValue
6:     for each node ∈ nodeSet do
7:         newSet ← nodeSet \ node
8:         tempCost ← cost(startNode, node) + recursiveTSP(node, newSet)
9:         if tempCost < cost then
10:            cost ← tempCost
11:            cheapestNode ← node
12:        end if
13:    end for
14:    store cost and cheapestNode for this subproblem
15:    return cost
16: end procedure
```

5 Summary & Outlook

We created a web application which allows users to easily create a custom round trip. As POIs we use Wikipedia articles received trough the extensive MediaWiki API. Many different future improvements are possible. We want to implement the possibility for the users to filter and search the received Wikipedia articles. For example if someone is only interested in visiting churches, it is possible to filter the articles for churches only or to search for specific ones. Furthermore, we also want to implement a navigation function, enabling users to fully enjoy the visited area. In fact, we already implemented a navigation function using the leaflet library by Liedmann[8], as can be seen in figure Figure 7. However, the library uses its own shortest path calculation algorithm and since it is mandatory for this project to develop them on our own, we did not include to navigation function yet.

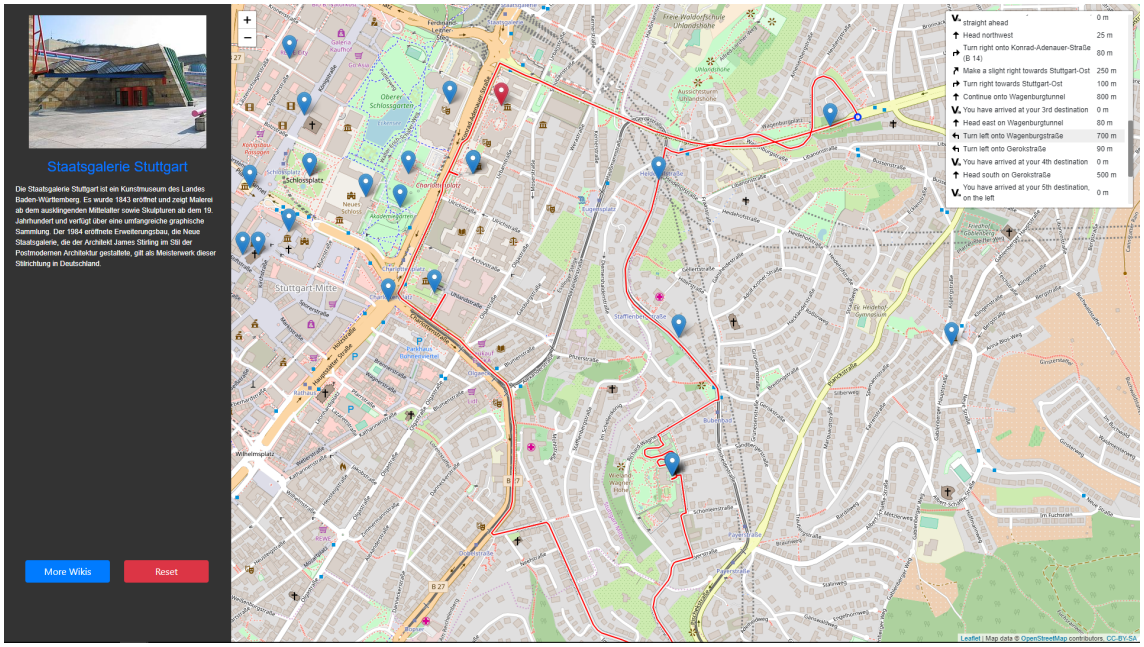

Figure 7: Wikipedia round trip planer with navigation.

[8] http://www.liedman.net/leaflet-routing-machine

References

[Bel57] Richard Ernest Bellman. Dynamic programming. 1957.

[Bel62] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9(1):61–63, 1962.

[Dij59] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[HK62] Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

[HW08] Mordechai Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008.