

Institut für Formale Methoden der Informatik

Universität Stuttgart  
Universitätsstraße 38  
D–70569 Stuttgart

Entwicklungsprojekt

## Trailstout

Aimn Ahmed, Samuel Holderbach, Dominik Krenz,  
Dominic Feldweg, Leonard Rupietta, Till Prölß, Ralf  
Baumann, Marcel Richter, Alexander Harner

**Studiengang:** Softwaretechnik

**Prüfer/in:** Prof. Dr. Funke

**Betreuer/in:** Claudio Proissl, M.Sc.  
Felix Weitbrecht, M.Sc.

**Beginn am:** 12. April 2022

**Beendet am:** 30. September 2022

## **Kurzfassung**

Im Rahmen eines Master Entwicklungsprojekts haben wir Trailscout entwickelt: eine Applikation zur automatisierten Wanderroutenerstellung als Maximierungsproblem. Wanderrouten sollen an schönen oder besonderen Orten vorbeiführen statt nur so schnell wie möglich von A nach B zu kommen. Aus OpenStreetMap Daten haben wir einen Straßengraph und Sehenswürdigkeiten extrahiert. Ein Nutzer legt in einem Web-GUI fest wo er wandern will und welche Sehenswürdigkeiten er interessant findet. Die Applikation berechnet anhand der Interessen einen Score für das besuchen der einzelnen Sehenswürdigkeiten. Für die Routenerstellung wird der Score, der in einer gewissen Zeit erreicht wird, maximiert. Durch die Umsetzung als Web-GUI mit zusätzlichen Server ist Trailscout auch auf mobilen Endgeräten nutzbar. Trailscout ist größtenteils mit Angular und Rust umgesetzt worden.

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>4</b>
<b>2 Daten</b>	<b>6</b>
2.1 OSM und FMI . . . . .	6
2.2 Graph Struktur . . . . .	6
2.3 OSM-PBF-Parser . . . . .	7
2.4 Graph Bibliothek . . . . .	11
2.5 Graph Konfiguration . . . . .	12
2.6 Ausblick . . . . .	13
<b>3 Algorithmen</b>	<b>14</b>
3.1 Problembeschreibung . . . . .	14
3.2 Algorithmen Bibliothek . . . . .	14
3.3 Vorverarbeitung der Daten . . . . .	15
3.4 Routenberechnung . . . . .	16
3.5 Benchmarks . . . . .	19
3.6 Zusammenfassung und Ausblick . . . . .	20
<b>4 Frontend</b>	<b>22</b>
4.1 Benutzung der Trailscout Webanwendung . . . . .	22
4.2 Architektur- und Designentscheidungen . . . . .	24
4.3 Ausblick für das Frontend . . . . .	28
<b>5 Server und Deployment</b>	<b>30</b>
5.1 Ngnix und Actix . . . . .	30
5.2 Docker . . . . .	30
5.3 Konfiguration . . . . .	31
<b>6 Zusammenfassung</b>	<b>32</b>
<b>Literaturverzeichnis</b>	<b>33</b>

# 1 Einleitung

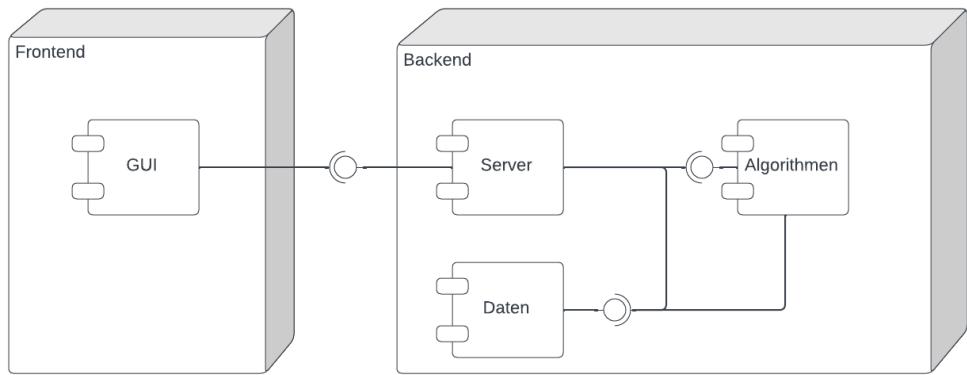
Eine Frage mit der jeder Tourist in einer fremden Stadt konfrontiert wird ist: was gibt es alles zu sehen und wohin gehe ich um das meiste aus meiner Zeit zu machen. Hierfür gibt es bereits viele Lösungen die bei der Lösung der Frage helfen. Touristenguides oder Sightseeingtouren bieten eine gute Möglichkeit um die wichtigsten Sehenswürdigkeiten einer Stadt zu besuchen. Diese Touren bieten jedoch nicht viel Zeit für das freie Erkunden der Städte, was für einige der Grund ist, warum sie von solche Touren abgeneigt sind. Es gibt also eine Entscheidende Zielgruppe die nach eigenem Belieben Städte erkunden möchten ohne dabei ziellos herumzuwandern.

Das Ziel dieses Projektes war es dieses Problem anzugehen. Und zwar sollte ein Tool entwickelt werden, dass den Nutzern erlaubt über einen Internet-Browser auf ihrem Computer oder Smartphone eine optimale Sightseeingroute zu berechnen. Hierzu müssen natürlich diverse Faktoren beachtet werden. Die Zeit die der Nutzer zur Verfügung hat, seine Vorlieben sowie die Öffnungszeiten der Sehenswürdigkeiten und Orte die besucht werden können unter anderem die Qualität einer Route beeinflussen. Außerdem soll die Anwendung nicht nur für Sightseeingtouren genutzt werden können, denn es werden nicht nur Museen und Monamente beachtet, sondern auch Bars, Restaurants oder Picknickplätze und vieles mehr. So bleiben dem Nutzer viele Möglichkeiten eine individuelle Route zu erstellen.

Für dieses Projekt wurden die Entwickler in drei Teams aufgeteilt. Ein Team hat sich mit der Aufbereitung der Daten von Openstreetmap beschäftigt. Ein anderes Team hat sich mit dem Problem genauer auseinander gesetzt und hat die Algorithmen entwickelt, die für die Routenberechnung zuständig sind. Das letzte Team hat sich mit der Entwicklung der Webanwendung auseinandergesetzt um eine Benutzeroberfläche zu bieten die leicht zu bedienen ist.

Abbildung 1.1 zeigt die grobe Architektur der Anwendung. Diese hat viele Ähnlichkeiten zu der Architektur einer herkömmlichen Webanwendung. Die Benutzeroberfläche auf dem Client ist über eine REST-API mit dem Server verbunden. Die API kann die Routenberechnung starten sowie über eine Datei-basierte Datenbank auf das Straßennetz und eine Sammlung von Sehenswürdigkeiten.

Im weiteren Verlauf dieser Arbeit wird in Abschnitt 2 darüber berichtet wie die Daten für das Problem aufbereitet wurden. Daraufhin wird in Abschnitt 3 besprochen, welche Algorithmen für die Routenberechnung verwendet wurden und wie sie funktionieren. Abschnitt 4 beschreibt die Benutzung der entwickelten Webanwendung und die genauen Architektur und Designentscheidungen die im Frontend getroffen werden mussten. Anschließend wird in Abschnitt 5 auf die Technologien eingegangen die für das Deployment der Anwendung relevant ist. Zuletzt bietet Abschnitt 6 eine kurze Zusammenfassung dieser Arbeit.



**Abbildung 1.1:** Komponenten von Trailscout

## 2 Daten

Dieses Kapitel befasst sich mit der Beschaffung und Aufbereitung der Daten, deren Ziel es ist eine passende Graph Bibliothek für das zugrundeliegende Problem zur Verfügung zu stellen. Hierfür wurden OpenStreetMap-Daten (OSM-Daten) [OSM] verwendet. Jegliche Schritte zur Erstellung des Graphen, sowie dessen Struktur und Aufbau werden erläutert. Außerdem werden alle Funktionalitäten der Graph Bibliothek dargelegt. Da die OSM-Rohdaten sehr detailreich und breit gefächert sein können, wurde auf die Konfigurierbarkeit des Graphen viel Wert gelegt. Dadurch kann der Nutzer die Daten weitestgehend an seine Präferenzen anpassen.

### 2.1 OSM und FMI

Für die Generierung von Wanderrouten auf einem Straßengraphen, welche möglichst viele Sehenswürdigkeiten enthalten, sind entsprechende Daten notwendig. Hierfür wurden OSM-Daten verwendet.

Als 2004 in Großbritannien das OSM-Projekt ins Leben gerufen wurde, war das Ziel weltweite, freie Kartendaten zur Verfügung zu stellen. Zu diesen Kartendaten zählen zum Großteil die Verkehrsinfrastruktur (Straßen und Wege etc.), aber auch Gebäude, "Point of Interest", Flächennutzungsdaten, Küsten und Ländergrenzen. Die meisten Daten stammen von Freiwilligen unter Verwendung von GPS-Geräten. Speziell für OSM entwickelte Editoren wurden zur Speicherung in eine zentrale Datenbank verwendet. Aufgrund der variierenden Qualität der OSM-Daten, existieren sowohl detailreiche Gebiete, als auch Gebiete mit relativ wenigen Daten.

Auf der Geofabrik-Website werden regionale Extrakte der OSM-Rohdaten im PBF-Format kostenfrei zum Download angeboten. Da nicht alle Informationen in den OSM-Daten für unsere Problemstellung relevant sind, müssen diese Daten entsprechend aufbereitet und auf das wesentliche gekürzt werden. Das FMI-Format [FMI] stellt hingegen ein kompaktes Format für Straßengraphen dar. Dennoch beinhaltet das FMI-Format beispielsweise keine Sehenswürdigkeiten und muss daher erweitert werden. Um solch einen Straßengraphen in Rust realisieren zu können, gibt es diverse Bibliotheken, mit Hilfe derer man OSM-Rohdaten im PBF-Format einlesen und verwenden kann.

### 2.2 Graph Struktur

Bevor wir auf die Erstellung des Straßengraphen eingehen, wird zunächst die Struktur und der Aufbau beleuchtet. Hierbei haben wir uns stark an dem FMI-Format orientiert, da es auf effiziente Start-Ziel-Anfragen ausgelegt ist. Der gerichtete und gewichtete Graph besteht aus einer Knoten-,

Kanten- und Sehenswürdigkeitenliste. Jegliche Listen sind dabei statisch (Array), wodurch Zugriffe signifikant schneller ausgeführt werden können. Knoten bestehen aus einer ID und ihrem Längen- und Breitengrad.

Eine Kante stellt eine Verbindung zwischen zwei Knoten im Straßengraph dar. Sie enthält daher Start- und Zielknoten, wobei hier nur die ID der Knoten verwendet wird. Außerdem wird die Distanz zwischen diesen Knoten gespeichert. Aufgrund der verschiedenen Straßenarten (Fußgängerzonen, Verkehrsberuhigte Bereiche, Feld- und Waldwege usw.) in den OSM-Daten, wird auch diese in der Kante hinterlegt. Die Straßenart kann nachher verwendet werden, um beispielsweise bestimmte Straßenarten bei der Wander routengenerierung auszublenden.

Jede Sehenswürdigkeit ist gleichzeitig auch ein Knoten aus der Knotenliste und enthält daher die korrespondierende Knoten-ID. Aus diesem Grund wäre es zwar nicht notwendig die Längen- und Breitgrade mitzuspeichern, da man diese über die Knoten-ID und der Knotenliste ermitteln kann, jedoch war es aus Gründen der Effizienz von großer Bedeutung, um Beispielsweise alle Sehenswürdigkeiten in einem bestimmten Bereich ermitteln zu können. Aufgrunddessen wurden die Koordinaten für jede Sehenswürdigkeit mitgespeichert. Attribute wie Name, Öffnungszeiten und Aufenthaltszeit wurden ebenfalls aus den OSM-Daten mit aufgenommen, da diese für das Zeitbudget im Algorithmus benötigt werden. In dem OSM-Daten haben Knoten und Kanten (in OSM sogenannte Ways) sogenannte Tags. Diese Tags enthalten jegliche Informationen, welche zur Beschreibung und Zuordnung dienen. Mit Hilfe dieser Tags kann festgestellt werden, um was für eine Art von Sehenswürdigkeit es sich handelt. Da diese jedoch sehr umfangreich sind und sich auch teilweise überlappen, haben wir uns dazu entschlossen mehrere Tags einer Kategorie zuzuordnen. Auf diese Weise können beispielsweise sämtliche Tags zu sportlichen Aktivitäten zusammengefasst und übersichtlicher dargestellt werden. Aus diesem Grund bilden wir verschiedene Tags auf selbst definierte Kategorien ab und speichern diese jeweils für eine Sehenswürdigkeit ab. Das letzte Attribut bildet die Wikidata-ID. Diese ID ermöglicht es bei Bedarf weitere Informationen über eine Sehenswürdigkeit sich zu beschaffen.

Eine weitere wichtige Liste im Graph bilden die Offsets. Sie haben die Aufgabe Zugriffe auf die Kanten effizienter zu gestalten. Es ermöglicht einem den direkten Zugriff auf alle ausgehenden Kanten eines Knotens, ohne dabei die Knotenliste extra durchsuchen zu müssen. Um dies zu gewährleisten, ist die Knotenliste nach Startknoten sortiert.

Während in der Knotenliste der Index des Knotens gleich der ID des Knotens ist, wurde zur gewisse Funktionalitäten die Liste nach Breitengrad sortiert benötigt. Aufgrund der häufigen Verwendung und der daraus resultierenden Effizienz Verluste durch mehrmaliges kopieren und sortieren der Knotenliste, wurde entschieden die Knotenliste nach Breitengrad sortier gesondert abzuspeichern. Diese enthält jedoch nicht gesamten Knoten, sondern nur die korrespondierende Knoten-ID als Referenz.

## 2.3 OSM-PBF-Parser

In diesem Abschnitt folgt der gesamte Parsing-Prozess. Außerdem wurden Benchmarks dieses Prozesses, sowie zu allen komplexeren Teilschritten erstellt, bei der die Performanz und Skalierung mit Hilfe von drei verschieden großen Graphen evaluiert wird. Um mit den gegeben OSM-Rohdaten arbeiten zu können und aus ihnen einen Graphen erstellen zu können, bietet Rust eine Vielzahl an

Bibliotheken an. Wir entschieden uns für die *osmpbf* Bibliothek [Hof]. Mit Hilfe dieser Bibliothek ist es möglich die PBF-Datei einzulesen und in Rust auf die einzelnen Knoten etc. und deren Attribute zugreifen zu können.

### 2.3.1 Parsing-Prozess

Im ersten Schritt des Parsing-Prozess werden mit Hilfe der *osmpbf* Bibliothek die OSM-Rohdaten aus der PBF-Datei eingelesen. Der Prozess des Einlesens wurde parallelisiert, wodurch eine massive Zeitsparnis erzielt wurde. Da jedoch die OSM-ID für die Knoten auf unsere eigenen Knoten-ID (0 bis Anzahl Knoten - 1) abgebildet werden müssen, ist dies nicht mehr sequentiell möglich, da alle Knoten parallel eingelesen werden. Aufgrund dessen wurde das Abbilden der OSM-IDs auf unsere selbst definierten Knoten-IDs auf einen späteren Zeitpunkt im Prozess verschoben. Anhand der OSM-Tags und einer Konfigurationsdatei wird dann geprüft, ob es sich um eine Sehenswürdigkeit handelt und in welche Kategorie diese gehört. Ungeachtet dessen, ob es sich um eine Sehenswürdigkeit handelt oder nicht, wird ein Knoten erstellt und in der Knotenliste gespeichert. Parallel dazu werden auch Kanten eingelesen. Während OSM mit Ways arbeitet, welche eine Aneinanderreihung von OSM-IDs von Knoten darstellt, verwenden wir einzelne gerichtete und gewichtete Kanten zwischen zwei Knoten. Daher musste man durch einen Way iterieren und diesen in Kanten aufteilen. Dabei muss darauf geachtet werden, dass für jede Hinkante und auch eine Rückkante erstellt werden muss. Es kann passieren, dass dadurch Kanten danach mehrfach abgespeichert werden mit gleicher oder auch unterschiedlicher Distanz. Dies wird zu einem späteren Zeitpunkt behandelt. Mit Hilfe einer weiteren Konfigurationsdatei für Kanten wird sichergestellt, dass nur Ways von bestimmten Arten betrachtet werden.

Nachdem nun alle Knoten, Kanten und Sehenswürdigkeiten erstellt wurden, werden im nächsten Schritt alle Sehenswürdigkeit nochmal genauer betrachtet. Einige der in OSM vorhandenen Sehenswürdigkeiten haben nur sehr minimale Informationen. Für uns ist dies ein Zeichen dafür, dass die Sehenswürdigkeit dementsprechend nicht aussagekräftig genug ist und es sich daher nicht lohnt diese in den Graph aufzunehmen. Dementsprechend ignorieren wir alle Sehenswürdigkeiten ohne Name. Die Kategorien Natur und Picknick bilden eine Ausnahme. Mitglieder dieser Kategorie haben oftmals keinen Namen, können aber dennoch interessant sein. Daher übernehmen wir diese mit einem selbst generierten Alternativnamen, bestehend aus der jeweiligen Kategorie zusammen und einer Nummerierung, welche zur Unterscheidung dient, in den Graphen.

Die OSM-Daten enthalten auch Gebäude, Bäume etc. Dadurch kann es passieren, dass Knoten ohne Kanten existieren. Diese werden in diesem Schritt aus den gespeicherten Knoten entfernt. Hierbei wird auch darauf geachtet, dass Sehenswürdigkeiten ebenfalls nicht zwingend Kanten besitzen. Sie werden natürlich beibehalten.

Zu diesem Zeitpunkt ist die Knotenliste soweit es geht vollständig. Das vorangegangen Problem bezüglich der Abbildung der OSM-IDs auf die Knoten-IDs wird in diesem Schritt angegangen. Jegliche OSM-IDs werden nun durch die korrekten Knoten-IDs in den Listen entsprechend ersetzt. Mehrfach vorkommen von Knoten werden ebenfalls beseitigt.

Wie bereits angesprochen kann es vorkommen, dass Sehenswürdigkeiten keine Kante besitzen und daher nicht erreichbar sind. Da es aber für die Wander routengenerierung zwingend notwendig ist, müssen auch diese Knoten erreichbar sein. Für jede Sehenswürdigkeit wird der nächstgelegene Knoten bestimmt und es wird eine Hin- und Rückkante zu diesem hinzugefügt. Dabei muss darauf geachtet werden, dass der nächstgelegene Knoten keine Sehenswürdigkeit ist.

Nachdem nun auch die Kantenliste weitestgehend vervollständigt wurde, kann man nun mit dem kürzen von unnötigen Kanten beginnen. Durch das Hinzufügen von Hin- und Rückkanten beim Einlesen der Ways, kann es durchaus passieren, dass Kanten nun mehrfach mit gleicher oder unterschiedlicher Distanz in der Liste existieren. Um diese zu entfernen, muss zunächst die Kantenliste nach Startknoten, dann nach Zielknoten und dann nach Distanz sortiert werden. Nun ist es möglich beim letzten Element beginnend durch die Liste zu iterieren und dabei immer die zwei benachbarten Kanten in der Liste zu vergleichen. Sollten zwei Kanten gleich sein (Start- und Zielknoten sind gleich), so entfernt man die untere Kante, da diese die höhere Distanz hat und man immer die niedrigste Distanz behalten möchte. Das Entfernen der Kanten geschieht über einen sogenannten *swap\_remove* aus der Rust Bibliothek. Dieser tauscht das zu löschen Element mit dem letzten Element aus der Liste. Dadurch ist zwar die Sortierung der Liste nicht mehr korrekt, jedoch spart man sich eine Menge Zeit bei der Berechnung ein. Im Anschluss muss daher die Liste nochmal neu sortiert werden.

Nachdem die Knoten und Kanten fertiggestellt wurden, kommen wir nun zu den Sehenswürdigkeiten. Die Sehenswürdigkeiten werden nach ihrem Breitengrad sortiert abgespeichert. Das ermöglicht die Suche nach Sehenswürdigkeiten um einen bestimmten Punkt effizienter durchzuführen. Anstatt über alle iterieren zu müssen, um zu überprüfen ob sie im gesuchten Bereich sind, kann der Bereich nach dem minimalen und maximalen Breitengrad eingegrenzt werden.

Sehenswürdigkeiten wie Picknickplätze und Grillplätze liegen oft sehr nahe in Gruppen beieinander. So sind zum Beispiel die einzelnen Grillstellen auf OSM ihre eigenen Sehenswürdigkeiten. Wir fassen solche Gruppen zu einer Sehenswürdigkeit zusammen um die Anzahl an Sehenswürdigkeiten zu reduzieren und die Routenqualität zu erhöhen. Sonst kann es vorkommen das Routen sehr viele Grillstellen beinhalten die sich alle am gleichen Platz befinden.

Im letzten Schritt, werden diese gewonnen und aufgearbeiteten OSM-Daten in einem passenden Format abgespeichert. Aus Gründen der Effizienz und Speicherverwaltung entschieden wir uns dazu, die Daten in einer Binärdatei zu speichern, da es zum Einen der Einlese-Prozess verschnellert hat und auf der anderen Seite auch wenige Speicherplatz einnimmt.

Möchte man die Anwendung starten, so wird die Graph-Datei eingelesen und der entsprechende Graph daraus generiert. Nachdem der Graph eingelesen wurde, werden die Offsets für die Kanten berechnet. Im Anschluss daran werden mit Hilfe der Konfigurationsdatei für die Sehenswürdigkeiten noch entsprechende Attribute gesetzt. Dazu gehören die Öffnungszeiten und die Aufenthaltszeiten.

In OSM können Öffnungszeiten in einem Textformat angegeben werden. Damit die Algorithmen, zur Erstellung der Wanderrouten, diese Öffnungszeiten sinnvoll einbinden können, parsen wir den Text mittels der *opening-hours* Rust Bibliothek. Das resultierende Objekt kann einfach verwendet werden. Zum Beispiel ist es möglich per Funktionsaufruf abzufragen, ob zu einem gegebenen Zeitpunkt die Sehenswürdigkeit geöffnet ist oder wann diese schließt. Nicht alle Sehenswürdigkeiten haben eine angegebene Öffnungszeit in OSM und es kann vorkommen, dass die angegebene Öffnungszeit sich nicht an das Textformat hält. In solchen Fällen schlägt das parsen fehl. Bei fehlgeschlagenen Fällen verwenden wir benutzerdefinierte Öffnungszeiten, die wir in unserer Konfigurationsdatei

*sights\_config.json* für jede Kategorie definiert haben. Die Rust Bibliothek ist leider relativ neu (etwa 1 Jahr alt). Dementsprechend treten noch Fehler und unerwartetes Verhalten auf, welche dazu führen können, dass die Sehenswürdigkeit einen unklaren Öffnungsstatus hat. Unserer Erfahrung nach tritt dies relativ selten auf. Trotz dieser Nachteile ist die Bibliothek die bei weitem beste Lösung um Öffnungszeiten zu parsen. Es gibt keine Alternativen in Rust und etwas selbst zu schreiben hätte signifikanten Aufwand benötigt. Bei den Aufenthaltszeiten werden die Werte aus der Konfigurationsdatei den Sehenswürdigkeiten entsprechend zugewiesen.

Im letzten Schritt wird die Knotenliste, welche nach Breitengrad sortiert ist, aus der Knotenliste berechnet und gespeichert. Der Graph enthält nun alle nötigen Daten und kann für die Wanderroutengenerierung verwendet werden.

### Osmium Tool

Die bei Geofabrik verfügbaren Daten beinhalten viele Informationen die wir nicht brauchen. Um das parsen zu beschleunigen und die Größe der Ursprungsdateien zu reduzieren verwenden wir das Osmium Tool [TH]. Es ist eine einfache Commandline Anwendung die das rudimentäre filtern von OSM Daten erlaubt. Damit können wir viele unnötigen Daten die das OSM Format beinhaltet schnell und effizient entfernen. Z.B. können wir so die Größe der OSM Datei für Deutschland von fast 4GB auf knapp unter 1GB reduzieren. Diese Vorverarbeitung ist optional und vermindert nur die Zeit und RAM Anforderungen zum Graph parsen. Für mehrere GB große Graphen kann dies auf vielen Computern mit weniger guter Ausstattung durchaus relevant sein.

#### 2.3.2 Benchmarks

Es wurden einige Benchmarks durchgeführt um die Performance und den Speicher Verbrauch des Parsing Prozesses zu messen. Während der Entwicklung gab es immer wieder Probleme weil die 32 gb RAM des Servers nicht ausreichten um den Deutschland Graphen zu parsen. Diese Probleme wurden auf verschiedene Weise behoben, was sich in den Ergebnissen zeigt. Die Performance des Parsing Prozesses war keine oberste Priorität, wurde allerdings auch nicht vernachlässigt, denn ein schneller Ablauf hilft in der Entwicklung und beim Testen.

Der gesamte Prozess wurde in drei Schritten getestet, erst die Vorverarbeitung durch das Osmium Tool, dann der eigentliche Parsing Prozess und schließlich das laden des vorher fertiggestellten Graphen. Jeder test wurde auf Bremen, Baden-Würrtemberg und Detuschland ausgeführt.

Osmium	Zeit (s)	Speicher (Gb)	Parsing	Zeit (s)	Speicher (Gb)
Bremen	0,79	1,382620	Bremen	12,31	0,323032
BW	42,01	1,467612	BW	382,00	2,177512
DE	291,48	1,484068	DE	2783,82	13,768688

Laden	Zeit (s)	Speicher (Gb)
Bremen	0,17	0,032440
BW	3,29	1,224700
DE	27,05	7,829248

## 2.4 Graph Bibliothek

Die Graph Bibliothek enthält Funktionen um den Graph von einer zuvor erstellten fmibin Datei zu generieren. Um einen Graphen einzulesen wird die `parse_from_file()` Funktion mit dem Pfad zur Datei aufgerufen. Die Datei wird hierbei mithilfe der bincode Bibliothek eingelesen. Anschließend wird eine Offset Liste berechnet und eine nach Latitude sortierte Node Liste berechnet und dem Graph Objekt hinzugefügt. Schließlich wird das erstellte Graph Projekt als Ergebnis der Funktion zurückgegeben. Über das Graph Objekt können nun auf alle wichtige Funktionen und Felder des Graphen zugegriffen werden. Diese umfassen die Nodes des Graphen nach ihrer Id sortiert, eine Liste mit Node Ids nach Latitude sortiert, eine Liste mit Kanten, eine Liste mit Offsets und eine Liste mit Sehenswürdigkeiten. Außerdem gibt es Funktionen um die ausgehenden Kanten eines Node zu finden, um den nächsten Node zu einem bestimmten Punkt zu finden (`get-nearest-node`) und um alle Sehenswürdigkeiten in einem Gebiet zu identifizieren(`get-sights-in-area`). Zwei der Funktionen sind etwas komplizierter und werden in den folgenden Sktionen genauer beschrieben.

### 2.4.1 get-nearest-node

Diese Funktion findet den nächsten Knoten zu einer bestimmten Position. Für einen Menschen mit einer Karte ist diese Aufgabe normalerweise sehr einfach, für den Computer überraschenderweise allerdings nicht. Unsere erste naive Implementierung löste das Problem indem einfach über die gesamte Knoten Liste iteriert wurde um das beste Ergebnis zu finden. Diese Lösung hat für kleinere Graphen gut funktioniert, war aber bei großen Graphen wie Deutschland zu langsam. Daraufhin wurde eine deutlich schnellere binäre Suche implementiert. Die Knoten Liste wird beim einlesen nach Latitude vor sortiert und dann der zur Position nächste Knoten mithilfe der binären Suche identifiziert. Dieses erste Ergebnis ist natürlich nur nach Latitude optimal, nicht nach tatsächlicher Distanz. Von diesem Ergebnis wird nun die Liste in beide Richtungen durchgearbeitet und bessere Ergebnisse gespeichert. Die Differenz zwischen der Latitude der gesuchten Position und der Latitude der Knoten nimmt hierbei stetig zu, wodurch wir eine minimale Distanz berechnen können. Sobald diese minimale Distanz das bisher beste Ergebnis übersteigt, bricht der Prozess ab und gibt die gefundene Knoten-Id zurück. In Tests war diese neue Implementierung auf einem Bremen Graph um Faktor 500 schneller.

### 2.4.2 get-sights-in-area

Mit dieser Funktion werden alle Sehenswürdigkeiten in einem Radius berechnet. Die Laufzeit dieser Methode nimmt, ähnlich zu `get-nearest-node` mit steigender Node Anzahl zu, weshalb eine naive Implementierung für unsere Zwecke nicht ausreichte. Die Sehenswürdigkeiten werden bereits während des Parsing Prozesses nach Latitude sortiert, weshalb wir hier ohne weiteres eine binäre Suche anwenden können. Mithilfe der binären Suche werden eine untere und obere Schranke identifiziert und ein Slice der gesamt Sehenswürdigkeiten extrahiert. Über diesen Slice wird anschließend iteriert und die Distanz jeder Sehenswürdigkeit zur Ziel-Position berechnet und unpassende Elemente entfernt. Der fertig bearbeitete Slice wird anschließend zurückgegeben.

## 2.5 Graph Konfiguration

Der Graph in den OMS Daten ist sehr umfangreich. Um trotzdem einen Überblick zu haben, haben die verschiedenen Knoten verschiedene Tags, die diese Knoten verschiedenen Kategorien zuordnet. Das selbe gilt auch für die Kanten, welche beispielsweise als Autobahnkante gekennzeichnet sind. Durch diese Kategorien oder auch Tags, können die verschiedenen Knoten und Kanten gefiltert und dadurch gezielter weiterverarbeitet werden. Da für eine Wander routenerstellung nur Kanten (und damit Wege) von Interesse sind, die auch zu Fuß verwendet werden können (also beispielsweise keine Autobahnen), mussten die vielen verschiedenen Kantentypen herausgefunden werden und die für die Routen interessanten Kantentypen ausgewählt werden. Das gleiche musste bei den noch umfangreichereren Tags für die Knoten durchgeführt werden. Da diese Daten grundlegend für eine gute Routenerstellung sind, musste hier eine ziemlich aufwendige Recherche durchgeführt werden. Um auch später Änderungen an den verschiedenen Tags der Knoten und der Kanten zu vereinfachen, wurde die genaue Definition der verschiedenen Tags in ein config file ausgelagert. Dadurch muss nur an dieser Stelle eine Änderung vorgenommen werden, falls neue Tags inkludiert werden sollen oder alte entfernt werden sollen. Für die Entwicklung hatte dies auch den Vorteil, dass es für uns einfacher war, mit verschiedenen Tags Tests durchzuführen und diese in der Applikation auszuprobieren und die Ergebnisse direkt anzuschauen.

### 2.5.1 Sehenswürdigkeiten

Bei den Sehenswürdigkeiten die Aufteilung der Kategorien und die zugehörigen Tags. und das config file Um die Sehenswürdigkeiten übersichtlicher darstellen zu können, werden für diese verschiedene Überkategorien erstellt. Diese sind Aktivitäten, Bademöglichkeiten, Grillplätze/Picnic Spots, Museen/Ausstellungen, Natur, Nachtleben, Restaurants, Sehenswürdigkeiten, Shopping und Animals.

Unter Aktivitäten werden Freizeitparks, sowie sportliche Aktivitäten, wie ice-skating, surfing, und so weiter zusammengefasst. Als Bademöglichkeiten werden Wasserparks, Schwimmhallen, sowie allgemeine Schwimmareale bezeichnet. Unter Museen/Ausstellungen werden Gallerien, Planetarien und Museen zusammengefasst. Für Seen, Flüsse, Parks, Lagunen und Aussichtspunkte, wird der Überbegriff Natur verwendet, während Pubs, bars und Nachtclubs als Nachtleben zählen. Unter Sehenswürdigkeiten werden historische Gebäude (Gebäude, Ruinen, ...) und touristische Punkte (Obelisken, Kunstwerke, ...) vereinigt. Mit dem Überbegriff shopping werden shoppingmalls zusammengefasst. Als Animals werden Zoos und Aquarien bezeichnet.

### 2.5.2 Kantentypen / Kantenarten

Mit Hilfe der Kantentypen können die verschiedenen Wegetypen herausgefiltert werden. Bei der Entscheidung, welche Typen inkludiert werden, wurde darauf geachtet, welche Kanten zu Fuß verwendet werden können und welche definitiv nicht zu Fuß verwendet werden können (zum Beispiel Autobahnen und Schnellstraßen). Es wurden Straßen inkludiert, bei denn klar hervorgeht, dass hier Fußgänger laufen können. Ein Beispiel hierfür sind die verkehrsberuhigten Bereiche. Im Graph und damit in der Config Datei enthalten sind: unclassified (Öffentlich befahrbare Nebenstraßen), residential (Tempo-30-Zonen), service (Privatgelände), living\_street (Verkehrsberuhigter Bereich), pedestrian (Fußgängerzone), track (Wirtschafts-, Feld- oder Waldweg), road (Straße unbekannter

Klassifikation), footway (Gehweg), bridleway (Reitweg), steps (Treppen auf Fuß-/Wanderwegen), corridor (Ein Gang im Inneren eines Gebäudes), path (Wanderwege oder Trampelpfade), primary (Straßen von nationaler Bedeutung), secondary (Straßen von überregionaler Bedeutung), tertiary (Straßen, die Dörfer verbinden)

## 2.6 Ausblick

Die Daten von OpenStreetMap eignen sich gut zur Herstellung eines Straßengraphen, für Sehenswürdigkeiten aber leider nicht. Selbst eine strikte Auswahl an OSM Tags erlaubt es nicht wirklich interessante Orte von uninteressanten zu unterscheiden. Über OSM ist keine Wertung möglich. Zusätzliche Datenquellen wären dafür nötig. Zum Beispiel wäre ein Abgleich mit Google Daten denkbar. Über die Anzahl und Güte von Reviews könnten wirkliche Highlights oder viel besuchte Punkte identifiziert werden.

Es sollte einfach möglich sein Trailscout über Deutschland hinaus auszuweiten. So weit wir wissen gibt es keinen Grund wieso nicht auf die OSM Daten anderer Länder verwendet werden könnten. Nur die Qualität der OSM Daten ist entscheidend. Wenn das Kartenmaterial für andere Länder auch so vollständig ist wie für Deutschland wäre der Aufwand minimal. Lediglich die Konfigurationsdateien müssten geändert werden.

# 3 Algorithmen

Dieses Kapitel erklärt die Algorithmen, die wir für das Erstellen einer Wanderroute verwendet haben und beschreibt ihre konkrete Umsetzung innerhalb der Applikation.

## 3.1 Problembeschreibung

In Kapitel 2 wurden die Straßen- und Sehenswürdigkeitendaten bereits beschrieben, welche die Probleminstanzen für die Generierung von Wanderrouten in Trailscout darstellen. Das Ziel ist es, anhand dieser Daten Wanderrouten zu generieren, die möglichst viele dieser Sehenswürdigkeiten in einer festgelegten Zeit besucht. In der Theorie ist dieses Maximierungsproblem auch als Orienteering Problem (OP) bekannt.

Das Orientierungsproblem (OP) ist wie folgt definiert: Sei  $N = \{n_1, \dots, n_{|N|}\}$  eine Menge von Knoten in einem zusammenhängenden Graph, so dass jeder Knoten  $n_i \in N$  einen Wert  $s_{n_i}$  hat. Das Ziel des OP ist es, eine Menge von Knoten  $N^* \subseteq N$  so zu besuchen, dass die Punktzahl der besuchten Knoten maximiert und ein vorgegebenes Zeitlimit eingehalten wird. In unserem Fall entsprechen die Knoten den Sehenswürdigkeiten und das Zeitlimit der maximal angegebenen Dauer der Wanderung.

Neben dem OP beschäftigt sich auch das Tourist Trip Design Problem (TTDP) mit dem Generieren von Routen entlang von Touristenattraktionen. Bei der Recherche nach Algorithmen für Trailscout haben wir uns sowohl Algorithmen angeschaut, die das OP bzw. verschiedene Definitionen des OP lösen, als auch Algorithmen für das TTDP.

## 3.2 Algorithmen Bibliothek

Trailscout beinhaltet eine Algorithmen Bibliothek, die für das Berechnen von Wanderrouten zuständig und auf die Benutzung innerhalb des Trailscout Services zugeschnitten ist. Das bedeutet, dass die Bibliothek die Metadaten und Parameter für die Routenberechnung, die der Service vom Frontend erhält, verarbeiten, die Routenberechnung durchführen und anschließend eine Route mit allen nötigen Meta-Informationen ausgeben kann, welche daraufhin vom Frontend dargestellt werden können. Die Algorithmen Bibliothek wurde, wie auch die Bibliothek der Datenschicht und der Trailscout Service selbst, in *Rust* geschrieben. Im Moment beinhaltet die Bibliothek zwei Implementierungen von Algorithmen, die das Problem der Erstellung einer Wanderoute mit möglichst hoher Belohnung innerhalb des gegebenen Zeitbudgets zu lösen versuchen – einen deterministischen Greedy Algorithmus und einen heuristischen Simulated Annealing Algorithmus [LV12]. Für die Berechnung einer Wanderoute benötigt die Bibliothek – unabhängig von der verwendeten konkreten Algorithmus-Implementierung – die folgenden Parameter als Eingabe:

- Graph, auf dem die Berechnung stattfinden soll (siehe Kapitel 2)
- Startzeitpunkt der Route
- spätester Endzeitpunkt der Route
- Laufgeschwindigkeit
- Koordinaten des Startpunktes
- maximaler Radius, in dem Sehenswürdigkeiten betrachtet werden sollen
- Präferenzen für Sehenswürdigkeiten bzw. Kategorien von Sehenswürdigkeiten

Die Ausgabe der Route erfolgt als Sequenz mehrerer Sektoren, von denen jeder Sektor – mit Ausnahme des letzten Sektors – zu einer Sehenswürdigkeit führt. Der letzte Sektor führt zurück zum Startknoten der Route. Die berechneten Routen sind demnach ein Rundgang, der zum Beispiel am Hotel oder Wohnort des Benutzers startet und diesen anschließend wieder dorthin zurückleitet. Jeder Sektor beinhaltet die Sehenswürdigkeit, zu der dieser führt, sowie die Sequenz von Graphknoten, die den Weg zu der Sehenswürdigkeit beschreibt. Die Bibliothek kann beliebig um neue Implementierungen erweitert werden, die lediglich die vorgegebenen Schnittstellen implementieren müssen. Das Vorgehen der beiden implementierten Algorithmen wird in den folgenden Abschnitten näher beschrieben.

### 3.3 Vorverarbeitung der Daten

Bevor eine Wanderoute berechnet werden kann, müssen zunächst die der Schnittstelle übergebenen Daten so verarbeitet werden, wie der Algorithmus sie intern zur Durchführung der Routenberechnung benötigt. Dabei sind für verschiedene Algorithmus-Implementierungen die grundlegenden Schritte, die zum Teil mit Hilfe der Datenschnittstelle von Trailscout durchgeführt werden, wie das Berechnen des Startknotens, die Abfrage der Sehenswürdigkeiten und das Assoziieren mit Belohnungen, die selben. Allerdings können je nach Implementierung noch weitere Schritte anfallen, wie beispielsweise die Vorberechnung der Distanzen beim Simulated Annealing.

#### 3.3.1 Berechnen des Startknotens und Abfrage der Sehenswürdigkeiten

Die Algorithmen Bibliothek benötigt zur Berechnung einer Route einen Startknoten auf dem jeweiligen Graphen und alle Sehenswürdigkeiten innerhalb eines sinnvoll gewählten Radius um den Startpunkt herum. Diese Daten frägt sie über eine Schnittstelle von der Datenschicht ab, welche für die Bereitstellung von sinnvollen Straßen- und Sehenswürdigkeitendaten aus OSM verantwortlich ist. Der Startknoten ist der zu den vom Frontend übergebenen Koordinaten des Startpunktes nächstgelegene Graphknoten. Die Sehenswürdigkeiten werden in einem Radius ausgegeben, den die Algorithmen Bibliothek abhängig von der vom Frontend übergebenen Laufzeit und Laufgeschwindigkeit wählt. Konkret wird dieser so gewählt, dass der Durchmesser des daraus entstehenden Kreises, mit gegebener Laufgeschwindigkeit, in gegebener Laufzeit, genau ein Mal abgelaufen werden kann.

### 3.3.2 Assoziieren von Sehenswürdigkeiten mit Belohnungen

Die Vorbereitung des Algorithmus beinhaltet weiter das Assoziieren von Sehenswürdigkeiten mit Belohnungen bzw. Scores. Der Algorithmus erhält dazu Nutzerpräferenzen für Sehenswürdigkeiten bzw. Kategorien von Sehenswürdigkeiten auf einer Ordinalskala zwischen 1 und 5, wobei 1 die niedrigste Präferenz und 5 die höchste Präferenz ist. Die Abwesenheit einer Präferenz wird im Algorithmus als Zeichen dafür interpretiert, dass sie bei der Routenberechnung keine Rolle spielt und deshalb ignoriert werden kann. Die Präferenzen für die Kategorien fungieren dabei als Standardwert für die jeweilige Kategorie, können jedoch von den Präferenzen für einzelne Sehenswürdigkeiten überschrieben werden. Diese Nutzerpräferenzen werden dann innerhalb des Algorithmus auf Belohnungswerte abgebildet. Dies haben wir deshalb gemacht, da wir hohen Präferenzen, in Relationen zu niedrigen, ein noch höheres Gewicht geben wollten. Im Moment wird die Belohnung zu einer Präferenz  $p$  mit der Formel  $2^{p-1}$ ,  $p > 0$  berechnet. Die durch den Greedy Algorithmus berechneten Routen beinhalteten dadurch zwar quantitativ etwas weniger Sehenswürdigkeiten, wurden jedoch qualitativ besser.

### 3.3.3 Vorberechnen der Distanzen bei Simulated Annealing

Der Simulated Annealing Algorithmus führt zudem noch eine weitere Vorberechnung aus. Da der Algorithmus aufgrund seiner Beschaffenheit erheblich mehr Distanzanfragen machen muss, als der Greedy Algorithmus, werden bei ihm die Distanzen ausgehend von den Sehenswürdigkeiten und dem Startknoten vorberechnet. Hierzu werden von allen Sehenswürdigkeiten und dem Startknoten One-to-all-Dijkstra Anfragen innerhalb des Radius durchgeführt und die berechneten Distanzen werden vom Algorithmus temporär gespeichert.

## 3.4 Routenberechnung

Im Anschluss an die Vorverarbeitung der Daten kann die eigentliche Berechnung der Route gestartet werden. Hier liegen in der Regel die größten Unterschiede bei den Implementierungen, da verschiedene Algorithmen meist komplett unterschiedliche Ansätze verfolgen, um das Problem zu lösen. In diesem Abschnitt wird das Vorgehen des Greedy Algorithmus und des Simulated Annealing Algorithmus im Detail beschrieben. Darüber hinaus gehen wir auf Vor- und Nachteile der Algorithmen ein.

### 3.4.1 Greedy Algorithmus

Die Idee des Greedy Algorithmus, wie wir ihn in Trailscout implementiert haben, basiert grob auf der Arbeit von Wörndl, Hefele und Herzog «Recommending a sequence of interesting places for tourist trips» (2017) [WHH17]. Das Ziel des Algorithmus ist es, in jeder Iteration – ausgehend vom Startpunkt bzw. der zuletzt besuchten Sehenswürdigkeit – die Sehenswürdigkeit aus der Menge aller Sehenswürdigkeiten auszuwählen, die als nächstes besucht werden soll. Dies soll so lange durchgeführt werden, bis das Zeitbudget so weit aufgebraucht wurde, dass keine weitere Sehenswürdigkeit mehr besucht werden kann, allerdings noch genügend Zeitbudget übrig ist, um wieder zurück zum Startknoten zu gelangen. Zudem sollen Sehenswürdigkeiten nicht mehr als ein

mal besucht werden. Zum Bestimmen des nächsten Aufenthaltes bewertet der Algorithmus die verfügbaren Optionen nach ihrer Belohnung sowie ihrer Distanz zum zuletzt besuchten Knoten. Die Formel, nach der die Sehenswürdigkeiten geordnet werden, teilt die Belohnung der Sehenswürdigkeit  $s$  durch die Distanz zu eben dieser (ausgehend vom letzten Aufenthalt  $v_{last}$ ):  $\frac{score(s)}{max(dist(v_{last}, s), 1)}$ .

Bei der Berechnung der Wanderroute geht der Greedy Algorithmus also folgendermaßen vor: Zu Beginn werden Sehenswürdigkeiten herausgefiltert, welche keine Belohnung erhalten haben und somit für die Routenberechnung irrelevant sind. Im selben Zug werden alle unbesuchten Sehenswürdigkeiten in einer Datenstruktur gespeichert. Anschließend wird ein One-to-all-Dijkstra vom Startknoten aus innerhalb des selben Radius ausgeführt, in dem die von der Datenschicht angefragten Sehenswürdigkeiten liegen. Der Algorithmus iteriert dann so lange, bis das Zeitbudget so weit aufgebraucht ist, dass keine weitere Sehenswürdigkeit mehr der Route hinzugefügt werden kann. In jeder Iteration wird dabei ein One-to-all-Dijkstra ausgehend von dem der Route zuletzt hinzugefügten Knoten durchgeführt. Mit Hilfe der durch den Dijkstra berechneten Distanzen werden die unbesuchten Sehenswürdigkeiten nach der oben genannten Formel gerankt. Anschließend wird über die sortierte Datenstruktur iteriert und für jede Sehenswürdigkeit wird überprüft, ob diese noch dem Ende der jetzigen Route hinzugefügt werden kann, ohne dabei das festgelegte Zeitbudget zu überschreiten. Dabei wird betrachtet, wie viel Zeit benötigt wird, um vom letzten besuchten Knoten zu der Sehenswürdigkeit und anschließend zurück zum Startknoten der Route zu gelangen. Das Einbeziehen von Öffnungszeiten sowie der geschätzten Länge des Aufenthaltes an den Sehenswürdigkeiten wird in Abschnitt 3.4.3 genauer erläutert. Die Distanz von der zu besuchenden Sehenswürdigkeit zum Startknoten kann aus den zu Beginn berechneten Distanzen, ausgehend vom Startknoten, abgelesen werden, da die Kanten des zugrundeliegenden Graphen in beide Richtungen betrachtet werden. Nachdem eine Sehenswürdigkeit der Route hinzugefügt wurde, beginnt die nächste Iteration des Algorithmus. Wenn der Route keine weitere Sehenswürdigkeit hinzugefügt werden konnte, bricht der Algorithmus ab und gibt die Route zurück.

Mit dem Greedy Algorithmus lassen sich bereits akzeptable Wanderrouten in relativ kurzer Zeit generieren. Jedoch hat der Greedy Ansatz ein großes Problem: Er betrachtet zu keiner Zeit die Route als ganzes, sondern lediglich die nach seiner Metrik berechnete, nächstbeste Sehenswürdigkeit. Aus diesem Grund haben wir einen weiteren Algorithmus in das Trailscout Framework eingebunden und implementiert, der auf einem komplett anderen Ansatz basiert.

### 3.4.2 Simulated Annealing

Der Simulated Annealing Algorithmus betrachtet das Problem als Instanz des OP und versucht mit einem heuristischen Simulated Annealing Ansatz die optimale Lösung zu approximieren, d.h., die Gesamtbelohnung zu maximieren. Die zugrundeliegende Verfahrensweise ist die selbe wie bei dem Simulated Annealing Algorithmus von Lin und Yu aus ihrer Arbeit «A simulated annealing heuristic for the team orienteering problem» (2012), welcher Instanzen des Orienteering Problem with Time Windows (OPTW) lösen kann. Die Arbeit von Lin und Yu beinhaltet keine konkrete Implementierung, da diese natürlich stark davon abhängt, wie Probleminstanzen definiert sind. Im Fall von Trailscout bestehen Probleminstanzen aus Straßen- und Sehenswürdigkeitendaten aus OSM. Der Aufbau dieser Probleminstanzen wird in Kapitel 2 im Detail erklärt. Diese haben wir im Sinne des Algorithmus von Lin und Yu als Instanzen des OPTW aufgefasst, wobei das Berücksichtigen von Zeitfenstern bei der Routenerstellung, sowohl für den Greedy Algorithmus, als auch für den Simulated Annealing, in Abschnitt 3.4.3 genauer erläutert wird.

Das Vorgehen des Simulated Annealing, wie auch anderer heuristischer Verfahren, zeichnet sich dadurch aus, dass das Problem als Blackbox betrachtet wird. Oftmals werden Phänomene aus der Natur simuliert, um sich iterativ einer optimalen Lösung anzunähern. Im Fall des Simulated Annealing wird ein natürlicher Abkühlungsprozess simuliert, wie bei einem Stück Eisen das zum Beispiel zu einem Schwert verarbeitet werden soll. Zu Beginn ist das Eisen noch sehr heiß und deshalb leicht verformbar. Der Schmied kann es deshalb leicht in die gewünschte Form bringen und Fehler ausbessern. Je mehr Zeit vergeht, desto eher nimmt das Schwert Gestalt an; gleichzeitig wird es immer schwerer, die grobe Form des Schwertes noch zu verändern, bis es schließlich ganz fest ist. Analog dazu startet das Simulated Annealing mit einer zufälligen Startlösung, die dann durch lokale Operationen zufällig mutiert wird. Dabei ist zu Beginn die Wahrscheinlichkeit höher, schlechtere Lösungen zu akzeptieren um einem verfrühten lokalen Maximum zu entkommen. Je länger der Algorithmus jedoch läuft, desto geringer wird diese Wahrscheinlichkeit, bis schließlich ein Ergebnis akzeptiert wird, das der optimalen Lösung im besten Fall sehr Nahe kommt. Der Simulated Annealing Algorithmus wird durch die folgenden internen Parameter konfiguriert:

- $T_0$  – Initiale Temperatur für das Simulated Annealing  $\in (0, 1]$
- $B$  – Multiplikator für die Iterationen pro Wiederholung; wird mit der Anzahl an Sehenswürdigkeiten multipliziert
- $\alpha$  – Cooldown-Faktor, der am Ende einer Wiederholung auf die Temperatur angewendet wird
- $MAX\_TIME$  – Maximale erlaubte Berechnungsdauer; ist diese nach einer Wiederholung erreicht, bricht der Algorithmus ab und akzeptiert die beste gefundene Lösung
- $N\_NON\_IMPROVING$  – Maximale Anzahl an erlaubten Wiederholungen ohne Verbesserung der Lösung; wird ebenfalls als Abbruchkriterium nach jeder Wiederholung überprüft
- $MAX\_ITER\_PER\_TEMP$  – Dieser Parameter limitiert die Iterationen pro Wiederholung und wurde von uns eingeführt, da der Workload per Wiederholung bei einer hohen Anzahl von Sehenswürdigkeiten in der Praxis zu hoch wurde

Das konkrete Vorgehen des Simulated Annealings wird in [LV12] mit Hilfe eines Kontrollflussdiagramms detailliert beschrieben und ist deshalb nicht Teil dieser Arbeit. Jedoch beinhaltet das Kontrollflussdiagramm einen kleinen Fehler, den wir beim Implementieren des Algorithmus bemerkten und behoben haben. Lin und Yu überprüfen in ihrer Version beim Ablehnen einer schlechteren Lösung nicht, ob seit der letzten Temperaturanpassung schon  $B * n_s$  Iterationen durchgeführt wurden, wobei  $n_s$  die Anzahl der betrachteten Sehenswürdigkeiten darstellt. Dementsprechend wird in diesem Fall auch keines der Abbruchkriterien überprüft, was dazu führen kann, dass der Algorithmus niemals abbricht.

### 3.4.3 Berücksichtigen von Aufenthalts- und Öffnungszeiten

Um die implementierten Algorithmen in praktischen Szenarien wirklich nutzbar zu machen, war es nicht ausreichend das beschriebene Problem der Erstellung einer Wanderroute mit gleichzeitiger Belohnungsmaximierung als Instanz des OP zu betrachten. Lin und Yu [LV12] schreiben in ihrer Arbeit beispielsweise auch, ihr Algorithmus löse das OPTW, welches Sehenswürdigkeiten neben einer Belohnung auch noch ein Zeitfenster zuweist, so dass der Besuch der Sehenswürdigkeit innerhalb dieses Zeitfensters erfolgen muss. Jedoch kann der Algorithmus auch ohne Zeitfenster

implementiert werden, da diese bei der Akkumulation der Belohnungen auch ignoriert werden können. Da die Schnittstelle der Datenschicht zu Beginn keine Daten für Öffnungszeiten aus den OSM-Daten auslesen konnte, wurde dies erst im Nachhinein in die Implementierung des Simulated Annealing sowie in die des Greedy Algorithmus eingebaut.

Die Daten für Öffnungszeiten sind nicht immer verlässlich. Bei unklaren Öffnungszeiten haben wir uns dazu entschieden, anzunehmen, dass die entsprechende Sehenswürdigkeit geöffnet ist und besucht werden kann. Wenn eine Sehenswürdigkeit zum Zeitpunkt der Ankunft noch geschlossen ist, wird eine Wartezeit fällig, die vom Zeitbudget abgezogen wird. Dementsprechend kann auf einer Route mit vielen geschlossenen Sehenswürdigkeiten auch nur eine geringere Belohnung eingesammelt werden, da jedes Mal gewartet werden muss, bis die jeweiligen Sehenswürdigkeiten geöffnet haben. Der Simulated Annealing Algorithmus generiert deshalb, bei ausreichender Laufzeit, mit hoher Wahrscheinlichkeit eine Route mit geringen oder sogar gar keinen Wartezeiten, da dann mehr Sehenswürdigkeiten auf der Route besucht werden können, was eine höhere Belohnung zur Folge hat.

Neben Öffnungszeiten gibt die Datenschnittstelle auch eine geschätzte Aufenthaltszeit für jede Sehenswürdigkeit an. Diese wird ebenfalls in die Routenberechnung miteinbezogen. Der Aufenthalt an einer Sehenswürdigkeit kann selbstverständlich nicht über die Öffnungszeiten hinweg andauern, weshalb die angegebene Zeit nur die maximale Aufenthaltsdauer angibt, die bei der Sehenswürdigkeit verbracht werden soll, wenn diese nicht frühzeitig schließt oder zum Startpunkt zurückgekehrt werden muss.

### 3.5 Benchmarks

Um die für Trailscout implementierten Algorithmen auf ihre Performance zu testen und die Qualität der berechneten Routen zu vergleichen, haben wir einige Benchmarks durchgeführt. Die Algorithmen wurden für die Benchmarks teilweise mit fixen Argumenten aufgerufen. Zudem wurden drei verschiedene Einstellungen definiert, die jeweils mit einem Radius von 500 Metern bzw. einem Kilometer getestet wurden. Die festen Argumente waren wie folgt definiert:

- Startzeitpunkt der Route – 14 Uhr
- spätester Endzeitpunkt der Route
- Laufgeschwindigkeit
- Koordinaten des Startpunktes
- maximaler Radius, in dem Sehenswürdigkeiten betrachtet werden sollen
- Präferenzen für Sehenswürdigkeiten bzw. Kategorien von Sehenswürdigkeiten

	1		2		3	
Algorithmus	500m	1000m	500m	1000m	500m	1000m
Greedy						
SA						

**Tabelle 3.1:** Benchmarkergebnisse zum Vergleich der Algorithmen auf dem Baden-Württemberg Graphen

### 3.6 Zusammenfassung und Ausblick

Im Zuge des Projekts wurde eine Algorithmen Bibliothek implementiert, die in den Trailscout Service eingebunden wurde. Für diese Bibliothek wurden zwei konkrete Algorithmen implementiert, ein deterministischer Greedy Algorithmus und ein heuristischer Simulated Annealing Algorithmus. Der Greedy Algorithmus rankt Sehenswürdigkeiten nach ihrer Distanz zum zuletzt besuchten Knoten und fügt die beste Option der Route hinzu, so dass die daraus entstehende Route das Zeitbudget nicht verletzt. Das Simulated Annealing basiert auf dem Algorithmus von Lin und Yu [LV12], der unter anderem das OPTW löst. Mit dem Simulated Annealing werden zwar in der Regel Routen mit einer höheren Belohnung erzeugt, der Greedy Algorithmus ist dafür wesentlich effizienter und benötigt auch bei einer großen Anzahl an Sehenswürdigkeiten nicht viel Laufzeit um eine akzeptable Route zu berechnen.

Die Performance des Simulated Annealing könnte durch einige Optimierungen noch deutlich verbessert werden und somit könnten qualitativ gute Routen in akzeptabler Laufzeit generiert werden. Der größte Teil der Laufzeit des Simulated Annealing fällt zum einen auf das Vorberechnen der Distanzen und zum anderen auf die lokale Suche zum Ende jeder Wiederholung. Demnach könnte man die Laufzeit stark verbessern, wenn man Sehenswürdigkeiten mit einer niedrigen Belohnung von Anfang an ignorieren würde. Man könnte beispielsweise anhand der unter den verschiedenen Kategorien höchsten geschätzten Aufenthaltszeit abschätzen, wie viele Sehenswürdigkeiten mit dem gegebenen Zeitbudget im besten Fall besucht werden könnten. Damit könnte man dann eine Heuristik bauen, die festlegt, wie viele Sehenswürdigkeiten für eine bestimmte Ausführung des Algorithmus ignoriert werden könnten. Welche Sehenswürdigkeiten genau ignoriert werden sollen, müsste noch untersucht werden. Eine Idee wäre es, eine ähnliche Metrik wie im Greedy Algorithmus zu verwenden, die Sehenswürdigkeiten mit niedriger Belohnung und hoher Distanz zum Startknoten verwirft.

Eine weitere Performanceverbesserung für den Simulated Annealing könnte erreicht werden, in dem man die lokale Suche nicht nach jeder Wiederholung bzw. Temperaturanpassung durchführt, sondern lediglich ein Mal am Ende der Berechnung auf der besten Lösung ausführt, um diese nochmals zu verbessern. Die Verbesserung der Gesamtbelohnung durch die lokale Suche ist gemessen an der zusätzlichen Laufzeit oft relativ gering. Allerdings wurde dieser Sachverhalt noch nicht mit Hilfe eines Benchmarks genauer untersucht und es gab auch vereinzelte Fälle, in denen die lokale Suche nochmal eine starke Verbesserung der Gesamtbelohnung zur Folge hatte.

Auch der Greedy Algorithmus besitzt noch Potential zur Verbesserung und Erweiterung. So könnte man den Greedy Algorithmus so anpassen, dass er nicht nur die Distanzen zwischen dem zuletzt besuchten Knoten und den unbesuchten Sehenswürdigkeit in seine Metrik mit einfließen lässt,

sondern auch die Distanz zurück zum Startknoten. Dadurch könnte die Qualität der Routen verbessert werden. Für bessere Performance könnte man dann ein Abbruchkriterium in den One-to-all-Dijkstra ausgehend von dem zuletzt besuchten Knoten einbauen, so dass dieser abbricht, sobald bezüglich der Metrik des Algorithmus keine Verbesserung des bisher besten gefundenen Wertes mehr möglich ist. Dabei macht man sich zu Nutze, dass die Distanzen im Dijkstra Algorithmus, vom Startpunkt zu den aus der Priority Queue entfernten Knoten, monoton wachsen.

Um das Problem zu beheben, dass der Greedy Algorithmus nicht die Route als ganzes optimiert, könnte man zudem, statt lediglich einen One-to-all-Dijkstra ausgehend vom Ende der Route durchzuführen, um die nächste zu besuchende Sehenswürdigkeit zu finden, einen Many-to-all-Dijkstra ausgehend von allen Sehenswürdigkeiten auf der Route durchführen. Anschließend müsste man dann die Position ermitteln, an der die, bezüglich der Metrik, beste gefundene Sehenswürdigkeit in die Route eingefügt werden soll. Dadurch, dass in jedem Schritt die gesamte Route betrachtet würde, würde auch die Qualität der Route verbessert werden.

Die bisher vorgestellten Verbesserungsvorschläge bezogen sich in erster Linie auf die Performance der Algorithmen sowie die Routenqualität – gemessen an der Gesamtbelohnung. Indem Belohnungen nicht mehr fest wären, sondern abhängig von der tatsächlichen Dauer des Aufenthalts an einer Sehenswürdigkeit, könnten die Routen zudem noch praktikabler gemacht werden. Dadurch könnte sichergestellt werden, dass die Bedeutung der Gesamtbelohnung einer Route nicht durch viele sehr kurze Aufenthalte verfälscht wird. Die Algorithmen sind zwar so implementiert, dass die tatsächliche Aufenthaltszeit bezüglich der geschätzten Aufenthaltszeit maximiert wird; wenn eine Sehenswürdigkeit jedoch frühzeitig schließt oder das Zeitbudget knapp ist, kann es passieren, dass Sehenswürdigkeiten sehr kurz besucht werden, die eingesammelte Belohnung jedoch die selbe wie für einen «vollen» Besuch ist.

Durch die aufgeführten Änderungen an den Algorithmen könnte die gesamte User Experience mit Trailscout verbessert werden, da der Service in der Lage wäre, qualitativ gute Routen in kürzerer Zeit zu berechnen und zu kurze Aufenthalte an den Sehenswürdigkeiten fast gänzlich vermieden würden.

# 4 Frontend

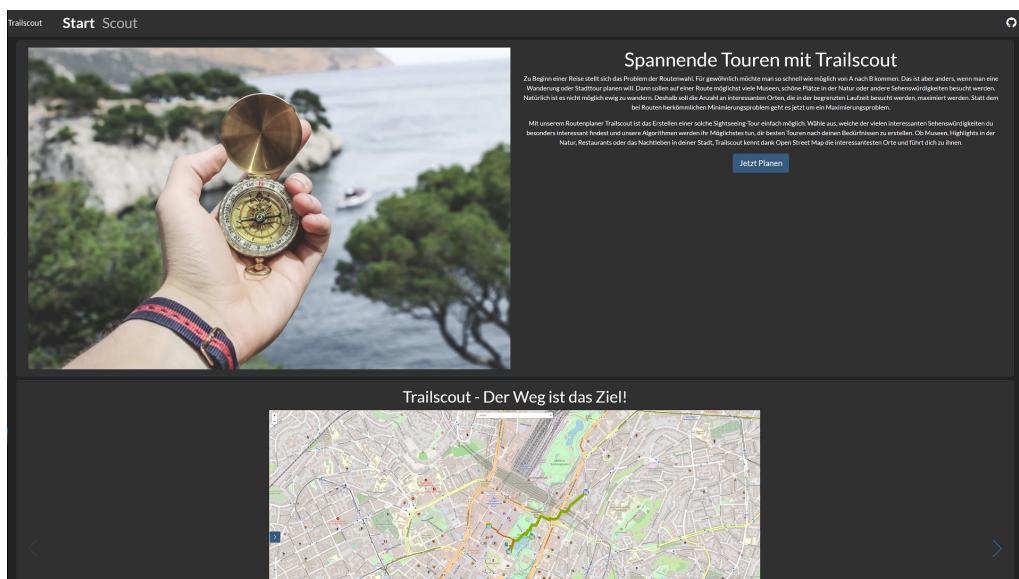
Dieser Abschnitt liefert einen tieferen Einblick in das Frontend der von uns entwickelten Trailscout Anwendung. Bei dem Frontend handelt es sich um eine Webanwendung die in Angular entwickelt wurde.

Ziel des Frontends der Anwendung soll es sein dem Benutzer die Möglichkeit zu geben eine Route zu planen. Hierfür gibt der User einen Punkt auf der Karte sowie einen Radius in dem sich die Sehenswürdigkeiten befinden sollen an. Neben weiteren Angaben soll der Benutzer in der Lage bestimmte Sehenswürdigkeiten aus der Route auszuschließen oder besonders hoch zu priorisieren. Diese gesammelten Daten sollen dann an das Backend gesendet werden, um das Ergebnis anschließend in der Webanwendung darstellen zu können.

## 4.1 Benutzung der Trailscout Webanwendung

Im Folgenden wird der genaue Ablauf bei der Benutzung der Trailscout-Anwendung beschrieben.

Navigiert der Benutzer auf die Trailscout-Homepage wird er zuerst von einer Einleitungsseite begrüßt, zu sehen in Abbildung 4.1. Hier erhält er eine Beschreibung, was genau das Ziel der Anwendung ist. Dort hat er die Möglichkeit auf die eigentliche Ansicht der Anwendung zu navigieren. Zuerst ist nur eine Karte zu sehen die von Leaflet bereitgestellt wird. Klickt der Nutzer auf einen

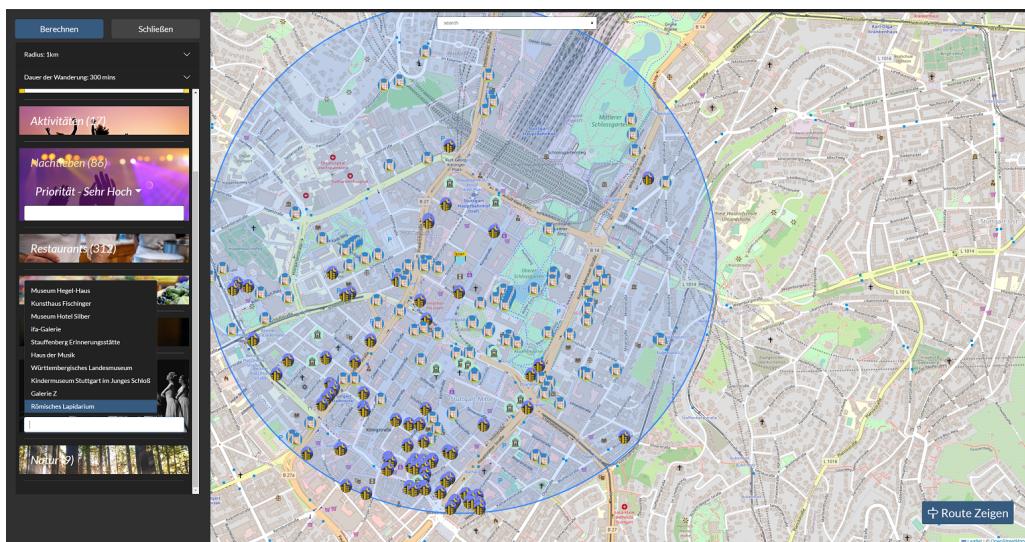


**Abbildung 4.1:** Trailscout Landing-Page

Punkt auf der Karte wird ein Marker gesetzt. Dieser Marker stellt den Startpunkt des Rundgangs dar. Ist ein Startpunkt gesetzt, kann links der Sidebar ausgeklappt werden.

Hier kann der User alle weiteren Einstellungen für den Rundgang festlegen. Eine weitere Einstellung die essentiell für den Rundgang ist, ist der Radius. Der Radius schränkt ein, in welchem Bereich Sehenswürdigkeiten auf der Karte dargestellt werden. Sind Radius und Startpunkt gesetzt, wird automatisch eine Anfrage an das Backend geschickt um die Sehenswürdigkeiten in dem Kreis zu aktualisieren. Der Nutzer wird diesbezüglich unten im Fenster informiert.

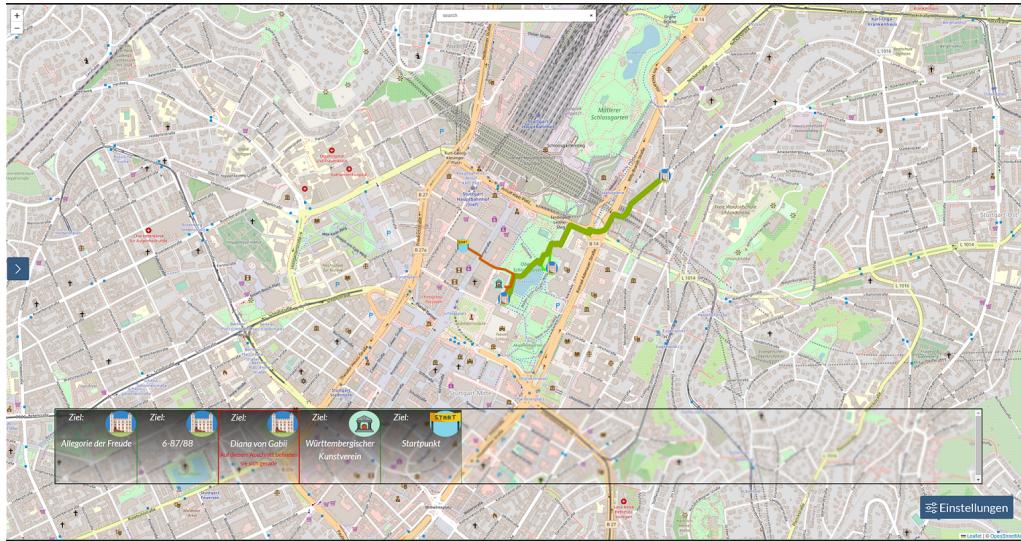
Im zweiten Abschnitt kann der User weitere Einstellungen für die Wanderung angeben. Oben kann die Laufgeschwindigkeit ausgewählt werden. Darunter kann die Start- und Endzeit der Wanderung angegeben werden. Zu beachten ist, dass die Wanderung nicht über 24 Stunden gehen kann, da die Eingabe eines Datums nicht unterstützt wird. Zusätzlich erhält der Nutzer die Information, dass seine Angaben zur Laufgeschwindigkeit und -zeit nicht zu dem gegebenen Radius passen, dieser also zu groß ist. Im unteren Abschnitt können die Sehenswürdigkeiten priorisiert werden. Dem



**Abbildung 4.2:** Trailscout Scoute-Seite in der man Einstellungen vornimmt um Ort, Zeit, und Präferenzen der Route zu wählen.

Nutzer werden eine Reihe von uns festgelegten Kategorien vorgestellt. Jeder der in Openstreetmap definierten Tags wurde einer Kategorie zugewiesen. Jede Kategorie ist durch einen eigenen Tab dargestellt, in dem auch angegeben ist, wie viele Sehenswürdigkeiten sich im Moment in dem Radius befinden. Klickt der Nutzer auf die Kategorie wird der Tab aufgeklappt. Gleichzeitig werden die Sehenswürdigkeiten auf der Karte dargestellt. Auf der Karte können die Sehenswürdigkeiten angeklickt werden, um den Namen sowie, falls vorhanden, ein Bild zu der Sehenswürdigkeit anzuzeigen. Im Sidebar kann die Kategorie an sich priorisiert werden. Standardmäßig ist noch keine Kategorie in die Wanderung miteingeschlossen. Außerdem kann im Suchfeld nach einzelnen Sehenswürdigkeiten aus der Kategorie gesucht werden. Dem Nutzer wird hier durch Vorschläge weiter geholfen. Durch Drücken der Enter-Taste wird der Tab erweitert und der User kann die Sehenswürdigkeit im einzelnen nochmal priorisieren. Ein Beispiel der Einstellungs-Ansicht ist in Abbildung 4.2 abgebildet.

Wurde mindestens eine Priorisierung abgegeben können die Infos für die Routenberechnung an das Backend gesendet werden. Sobald das Ergebnis angekommen ist, wechselt die Anwendung die Ansicht von den Einstellungen über zur Routenansicht. Über einen Knopf unten rechts kann zwischen den Ansichten gewechselt werden. Die Routenansicht versteckt den Radius und alle Sehenswürdigkeiten die nicht Teil der Route sind. Die Route ist auf der Karte zu sehen sowie eine Komponente, die die einzelnen Stationen auf der Route darstellt. Die Abschnitte können angeklickt werden um den Abschnitt genauer hervorzuheben. Der Sidebar beinhaltet nun eine Zusammenfassung des Requests der zu dieser Route geführt hat. So kann der User die generierte Route nochmal genauer überprüfen, um anschließend eventuelle Änderungen in den Einstellungen vorzunehmen. Die Routen-Ansicht ist in Abbildung 4.3 sichtbar.



**Abbildung 4.3:** Trailscout Scout-Seite mit der erhaltenen Route und dem dazugehörigen Routing-Tracker.

## 4.2 Architektur- und Designentscheidungen

Als Framework wurde Angular <https://angular.io/> gewählt. Gründe für diese Entscheidung waren die bereits vorhandene Erfahrung der Frontend Entwickler, die Auswahl an vielen externen Bibliotheken, die statisch deklarierte Programmiersprache Typescript, die dennoch dynamische Deklarierung erlaubt und somit eine schnelle und flexible Entwicklung gewährleistet, sowie die Flexibilität des Frameworks mit verschiedenen Bildschirmgrößen umzugehen. Alternative Frameworks waren React und Vue. Als externe Bibliotheken wurden verwendet:

- Bootstrap v5.1.3
- Bootstrap-Icons v1.8.3
- Bootswatch v5.1.3
- CookieConsent v3.1.1
- Leaflet v1.7.1

- Leaflet-Geosearch v3.6.0
- Ng-Block-Ui v3.0.2
- Ngx-cookie-service v13.2.1
- Ngx-cookieconsent v2.2.3
- swiper v8.3.2

Bootstrap bietet Widgets und Styles speziell entwickelt für Angular. Verwendete Icons wurden unter anderem aus der offiziellen Bootstrap-Icons Bibliothek verwendet. CookieConsent, Ngx-cookie-service, und ngx-cookieconsent erlauben das verwenden und den Hinweis auf Cookies für den User. Die verwendete Karte ist von Leaflet. Leaflet-Geosearch bietet eine Suchfunktion für Orte mit denen der Benutzer die Karte auf den gewünschten Ort zentrieren kann. Ng-Block-Ui wird verwendet um den Benutzer zu zeigen das die Route berechnet wird und dieser warten muss. Die Bildergalerie auf der Landing-Page ist von Swiper.

#### 4.2.1 Code Struktur

Der Code für das Frontend ist strukturiert in *App*, *Pages*, *Components*, *Data*, und *Services*. *App* ist die standard App-Komponente von Angular und dient als Einstiegspunkt. Sie verwaltet das Routing zu den *Pages*. Das *Pages* Modul enthält die verschiedenen Komponenten welche die jeweilige Single-Page Seite der Anwendung definieren und ihre Logik entählt. Im *Component* Modul befinden sich (möglichst) generische Komponenten die von den *Pages*-Komponenten benutzt werden. Dadurch entsteht eine lose Kopplung zwischen Komponenten und eine übersichtliche Baumstruktur entsteht. Zu den *Component*-Komponenten gehören z.B. die Karte, die Sidebar, die Elemente der Präferenzen, der Routen-Tracker, usw. *Data* enthält Klassen um die intern verwendeten Objekte statisch zu definieren. Und die *Services* enthalten Angular Services welche für den Datenaustausch mit dem Backend zuständig sind, für die Cookies, sowie Funktionen und Daten anbieten welche zwischen den Komponenten der Anwendung ausgetauscht werden müssen.

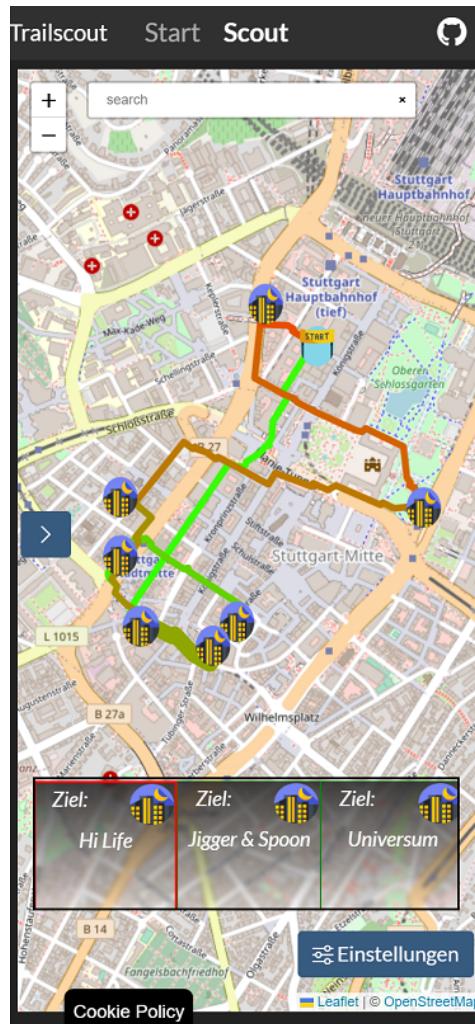
#### 4.2.2 Bilder und Icons

Die benutzten Bilder und Icons die in Trailscout benutzt werden, wurden von <https://www.pexels.com/de-de/> und <https://www.flaticon.com/de/> heruntergeladen.

#### 4.2.3 Design Philosophie

Da der Entwicklungszeitraum von 6 Monaten nicht viel Platz für Änderungen lies, wurde schon früh in der Entwicklung festgesetzt welches Prinzip das User Interface verfolgen soll um einen möglichst einfachen und benutzerfreundlichen Umgang für den Benutzer zu erlauben. Daher wurde die Mobile Version der Oberfläche parallel mit der Desktop Version entwickelt, was einem modernen *Mobile-First* Ansatz ähnelt. Das bedeutet: wenig Text, viele Bilder, Platz zum Erweitern anbieten. Da es eine sehr große Anzahl an verschiedenen Bildschirmgrößen gibt, war es ein wichtiger Ansatz der Oberfläche dynamisch zu skalieren. Damit die Anwendung auf kleinen Auflösungen genauso einfach zu bedienen ist wie auf großen mussten benutzbare Buttons möglichst groß sein, ohne

unnötig viel Platz einzunehmen. In Abbildung 4.4 sieht man die Mobile Version von Trailscout. In



**Abbildung 4.4:** Mobile Version der Trailscout Anwendung

dem Routing-Tracker erkennt man das die einzelnen Kacheln des Trackers durch ihre Größe leicht zu Treffen sind, jedoch eine Leichte Transparenz haben um die Karte im Hintergrund nicht zu viel wegzunehmen.

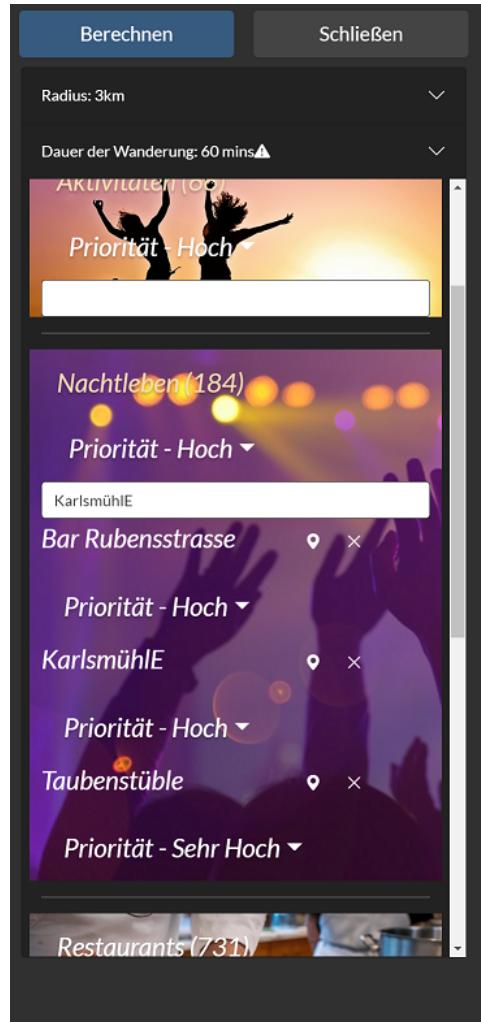
#### 4.2.4 Wichtige Features

Trailscout bietet eine Palette an nützlichen Features. Zu diesen Features gehören:

- Sehenswürdigkeiten Präferenzierung
- Routen-Tracker
- Bearbeitungs- und Routenmodus

## Sehenswürdigkeiten Präferenzierung

In Abbildung 4.5 wird ein Beispiel der Sehenswürdigkeiten Präferenzierung gezeigt. Es ist sowohl möglich die Kategorie selber eine Präferenz zu geben (von 'Gar nicht' bis hin zu 'Sehr Hoch') als auch einzelne Orte eine Präferenz zu geben. Diese Präferenzen werden zusammen mit anderen



**Abbildung 4.5:** Sehenswürdigkeiten Präferenz Komponente

Einstellungen an das Backend geschickt um eine optimale Wanderroute zu erhalten.

## Routen-Tracker

Hat man eine Route erhalten wechselt Trailscout in den Routenmodus. Der Routen-Tracker erscheint und erlaubt es dem Benutzer einen Überblick über die Route zu erhalten. Der Routen-Tracker ist in Abbildung 4.6 zu sehen. Die Route ist für jede enthaltene Sehenswürdigkeit in eine Sektion geteilt (plus eine für den Weg zum Startpunkt). Jede dieser Sektionen ist eine Kachel in dem Routen-Tracker.



**Abbildung 4.6:** Routen-Tracker von Trailscout

Beim hovern über der Kachel wird die dazugehörige Sektion der Route hervorgehoben. Beim Klicken auf die Kachel bewegt sich die Karte zu der Sektion. Durch diese Funktionen bekommt der Benutzer einen besseren Überblick über die Route.

### Bearbeitungs- und Routenmodus

Während der Benutzer eine Route auf der Karte hat und der Routen-Tracker sichtbar ist lassen sich keine Einstellungen mehr vornehmen. Beim Blick auf die Settings-Sidabar erhält man jediglich eine Zusammenfassung der eingetragenen Einstellungen die zu der aktuellen Route gehören. Um eine neue Route zu erhalten kann mit dem Klick auf den Einstellungs-Button zurück in den Bearbeitungsmodus gewechselt werden in dem nun Präferenzen, Zeiten und Radius bearbeitet werden können um eine neue optimale Wanderoute zu erhalten.

## 4.3 Ausblick für das Frontend

Durch die Zeiteinschränkung durch das Entwicklungsprojekt, waren die Frontend-Entwickler der Trailscout Anwendung nicht in der Lage jede Anforderung zu erfüllen. Im Folgenden werden Ideen und Verbesserungsvorschläge für das Frontend gegeben.

### 4.3.1 Pop-up Informationen der Sehenswürdigkeit

Bisher sind die Informationen die in den Popup Fenster dargestellt werden noch sehr knapp. Es wird der Name dargestellt und, falls vorhanden, ein Bild aus der Wikidata-Datenbank. Dieser Popup könnte erweitert werden um Informationen die in anderen Tags aus den OpenStreetMap-Daten widergespiegelt werden. Ein gutes Beispiel hierfür wären die Öffnungszeiten für die Sehenswürdigkeiten. Außerdem könnte der Nutzer darüber informiert werden, welche Tags die Sehenswürdigkeit zugeteilt bekommen hat, da diese mehrere haben könnten und lediglich einer von uns definierten Kategorie zugewiesen wurden.

Der Pop-up könnte auch, falls möglich, um weitere Funktionen erweitert werden. Im Moment ist es aufwendig eine einzelne Sehenswürdigkeit zu priorisieren die man gerade auf der Karte sieht. Im Moment muss sie angeklickt werden, um den Namen hervorzuheben, und anschließend im Suchfeld der Kategorie eingegeben werden. Hierfür würde ein Button in dem Pop-Up helfen, der die Sehenswürdigkeit direkt in die Liste der einzeln priorisierten Sehenswürdigkeiten hinzufügt. Dies kann speziell die Benutzung der Anwendung im mobilen Browser beschleunigen.

### 4.3.2 User Feedback

Das User Feedback kann ebenfalls an einigen Stellen erweitert werden. Hierfür können Pop-Ups, Tooltips oder Dialog-Fenster von Bootstrap nativ benutzt werden. Die Toasts am unteren Ende des Bildschirm können aber auch dafür verwendet werden um Informationen an den User zu vermitteln.

Falls der Nutzer eine Fehlermeldung bei der Pfadberechnung bekommt, kann dies mehrere Gründe haben. Wenn das Backend ausführliches Fehlermapping betreibt und dem Frontend genauer Angeben kann, ob der Fehler in den Einstellungen liegt, kann das Frontend wiederum genauer auf die Probleme hinweisen. Zum Beispiel kann der Fehlerhafte Input hervorgehoben werden.

# 5 Server und Deployment

Wir verwenden für Trailscout zwei verschiedene Web Technologien um den Nutzer die Verwendung von allen Trailscout-Funktionalitäten zu ermöglichen. Im folgenden Kapitel stellen wir diese vor und wie die Trailscout Applikation zu Konfigurieren ist. Eine Anleitung zur Installation findet sich im Projekt Wiki und wir nicht extra aufgeführt.

## 5.1 Ngnix und Actix

Um die Angular Applikation zu hosten setzen wir Ngnix [F5] ein. Dies ist ein weit verbreiteter Open-Source Webserver. Die Unterschiede unter den verschiedenen Webservers wie z.B. Apache sind für unser Projekt irrelevant, da alle statische Dateien ausliefern können. Performance, Konfigurierbarkeit und andere Faktoren waren für Trailscout im Rahmen dieses Projektes irrelevant. Unsere Wahl fiel aus Ngnix da wir es einfach ins Projekt einbinden konnten.

Um die Kommunikation mit den in Rust programmierten Backend Komponenten zu ermöglichen liegt es nahe, ein API mit einem Rust Web-Framework zu verwenden. Actix [The] ist ein einfach zu verwendendes und für Rust-Verhältnisse relativ ausgereiftes Web-Framework. Mit Actix haben wir einen einfachen Server implementiert der ein HTTP API bereitstellt um Sehenswürdigkeiten und Routen Anfragen von Frontend an den entsprechenden Algorithmus im Backend weiterzuleiten und das Ergebnis zurückzuschicken. Im Actix Servers stellen wir auch sicher, dass eventuelle Fehlermeldungen der Algorithmen angemessen aufbereitet werden bevor sie an das Frontend weitergeleitet werden. So vermeiden wir dem Nutzer zu technische Fehlermeldungen anzuzeigen.

## 5.2 Docker

Wir entschieden uns Docker [Doc] für das Projekt Deployment zu verwenden. Docker ermöglicht es Trailscout einfach und Plattform-unabhängig bereitzustellen. Lediglich Docker muss installiert sein, um Trailscout zu verwenden. Frontend und Backend sind in ihre eigenen Images getrennt. Diese Trennung hat mehrere Vorteile. Im jedem Container kann so die Umgebung auf die Komponente angepasst werden ohne Rücksicht auf die andere nehmen zu müssen. Außerdem könnte so einfach eine Skalierung von Trailscout umgesetzt werden, sollte es relevant werden. Front- und Backend könnten je nach Leistungsanforderungen auf unterschiedlicher Hardware laufen oder mehrere Backend Container die Anfragen eines Frontend Containers verarbeiten.

Der Aufbau beider Images ist ähnlich. Im ersten Schritt wird auf einen Build-System die Angular bzw. Rust Anwendung kompiliert. Im zweiten Schritt wird die kompilierte Anwendung auf das Linux System kopiert in dem die Komponente dann ausgeführt wird. Die Trennung von kompilieren und bereitstellen macht die Images deutlich kleiner, da sich die zur Kompilierung benötigten Ressourcen aus dem ersten Schritt nicht im finalen Image finden.

Das starten der Container erfolgt über ein Docker compose. Damit lassen sich mehrere Images die als eine Applikation zusammenarbeiten instantiiieren. Für Trailscout ermöglicht es, dass beide Komponenten im gleichen virtuellen Netzwerk sind und schnell so einfach miteinander kommunizieren können.

## 5.3 Konfiguration

Trailscout ermöglicht im Backend einige Konfigurationsmöglichkeiten. Um die Änderung der Konfigurationsdateien zu ermöglichen ohne die Images neu zu erstellen verwenden wir ein Docker Volume. Das ist im Prinzip ein geteilter Ordner zwischen den Containern und dem Host. Darin werden die Graph- und andere Konfigurationsdateien ausgelagert.

### 5.3.1 Konfig-Dateien

Im *config.json* können einige grundlegende Eigenschaften für die Backend Komponente festgelegt werden. Der verwendete Routing-Algorithmus und die Namen der zu verwendenden Graphen sind die wichtigsten Einstellungen. Für eine vollständige Liste verweisen wir auf das Projekt Wiki auf Github. Einstellung für das parsen der OSM Dateien finden sich in *sight\_config.json* und *edge\_type\_config.json*. Dort wird festgelegt welche Typen von Sehenswürdigkeiten und Straßen aus den OSM Daten in unseren Graphen übernommen werden.

### 5.3.2 Graphen

Im Volume, in dem Ordner *osm\_graphs* speichern wir die Graphen ab, die Trailscout zu Grunde liegen. Die Standardeinstellung ist den Graphen von Baden-Württemberg zu laden. Welcher Graph konkret geladen wird, ist in "config.json" festgelegt. Sollte man den Graphen wechseln wollen ändert man den *graph\_file\_path* Eintrag. Wenn man bis jetzt nur über das *osm.pbf* verfügt, also noch nicht den verarbeiteten Graphen, kann man zusätzlich den *source\_file* Eintrag auf die *osm.pbf* Datei ändern. Der Server generiert dann beim Start von selbst den Graphen, wenn er ihn nicht findet aber ein *source\_file* hat.

Optional kann das *preprocess\_osm.py* Python-Skript aus dem Trailscout Github (zu finden unter \backend) verwendet werden um automatisiert eine *osm.pbf* mittels Osmium zu filtern und in den Docker Volume zu bewegen. Das Skript funktioniert nur unter Linux.

## 6 Zusammenfassung

Die mit Angular entwickelte Web-GUI bietet eine nutzerfreundliche und einfache Oberfläche um eine optimale Wanderoute nach spezifischen Präferenzen zu erhalten. Nachdem man auf der Karte in Trailscout den gewünschten Startpunkt festgelegt hat gibt man Einstellungen ein die zur optimalen Wanderoute führen. Zu diesen Einstellungen gehören der Radius in dem man wandern möchte, die Start - und Endzeit, in Kombination mit Laufgeschwindigkeit, sowie die verschiedenen Präferenzen der Kategorien von Orten die man besichtigen oder vermeiden möchte. Im Anschluss erhält man dank dem Routen-Tracker eine Übersicht der Route. Trailscout unterstützt sowohl Desktop als auch Mobile Geräte.

Aus OpenStreetMap Daten erstellen wir einen Straßengraphen und extrahieren Sehenswürdigkeiten. Wir filtern Wege heraus auf denen sich nicht wandern lässt und grenzen die Sehenswürdigkeiten so weit ein wie es OSM erlaubt. Das Verwenden unterschiedlicher Graphen ist über eine Konfigurationsdatei einfach möglich. Für Trailscout verwenden wir hauptsächlich Graphen für Bremen, Baden-Württemberg und Deutschland.

Um eine passende Wanderoute zu erhalten haben wir zwei unterschiedliche Lösungsansätze implementiert. Einen deterministischer Greedy Algorithmus welcher grob auf [WHH17] basiert und ein heuristischer Simulated Annealing Algorithmus basierend auf [LV12]. Beide Algorithmen liefern eine passende Route unterscheiden sich aber in Effizienz und Qualität der Lösung.

# Literaturverzeichnis

- [Doc] Docker, Inc. *Docker*. <https://www.docker.com/>. (Accessed on 09/30/2022) (zitiert auf S. 30).
- [F5 ] F5, Inc. *nginx*. <https://www.nginx.com/>. (Accessed on 09/30/2022) (zitiert auf S. 30).
- [FMI] FMI. *Useful Stuff*. <https://fmi.uni-stuttgart.de/alg/research/stuff/>. (Accessed on 09/30/2022) (zitiert auf S. 6).
- [Hof] J. Hofmann. *GitHub - b-r-u/osmpbf: A Rust library for reading the OpenStreetMap PBF file format (\*.osm.pbf)*. <https://github.com/b-r-u/osmpbf>. (Accessed on 09/30/2022) (zitiert auf S. 8).
- [LV12] S.-W. Lin, F. Y. Vincent. „A simulated annealing heuristic for the team orienteering problem with time windows“. In: *European Journal of Operational Research* 217.1 (2012), S. 94–107 (zitiert auf S. 14, 18, 20, 32).
- [OSM] OSM. *GEOFABRIK // OpenStreetMap*. <https://www.geofabrik.de/de/geofabrik/openstreetmap.html>. (Accessed on 09/30/2022) (zitiert auf S. 6).
- [TH] J. Topf, S. Hoffmann. *Osmium Tool*. <https://osmcode.org/osmium-tool/>. (Accessed on 09/30/2022) (zitiert auf S. 10).
- [The] The Actix Team. *Actix*. <https://actix.rs/>. (Accessed on 09/30/2022) (zitiert auf S. 30).
- [WHH17] W. Wörndl, A. Hefele, D. Herzog. „Recommending a sequence of interesting places for tourist trips“. In: *Information Technology & Tourism* 17.1 (2017), S. 31–54 (zitiert auf S. 16, 32).