Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics: Games Engineering

A Recommender System for City Trip Planning

Alexander Hefele

Technische Universität
München

Department of Informatics

Bachelor's Thesis in Informatics: Games Engineering

A Recommender System for City Trip Planning

Ein Empfehlungssystem für die Planung von Städtereisen

Author: Alexander Hefele

Supervisor: Prof. Dr. Johann Schlichter

Advisor: Dr. Wolfgang Wörndl

Submission date: 17.08.2015

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

München, 17.08.2015

(Alexander Hefele)

Abstract

In this thesis we implemented a web-based recommender system for city trip planning which uses the Foursquare API to recommend interesting venues, like sights, bars, shops, and more on a route between two points. Users can specify a start and an end point within one city on a map, and select between a constraint-free and a constraint-based algorithm. The latter additionally takes the user's time and budget constraints into account. Then the system recommends, based on the user's interests, two routes between these points: one baseline approach and one representing our own implementation consisting of several improvements. Most notably, we incorporated Pearson's correlation coefficient to make the recommended places in the route correlate better with the user's preferences. We also improved the classification, as well as time and budget estimations for venues. On the website users had the option to rate the two paths based on several criteria. Analysis of the feedback showed that our implementation significantly improved how well the places in the path match the user's preferences. Users were overall more satisfied with our own algorithms compared to the ones of the baseline approach.

Inhaltsangabe

In dieser Arbeit haben wir ein web-basiertes Empfehlungssystem für die Planung von Städtereisen implementiert, welches die Foursquare API benutzt um interessante Plätze, wie z.B. Sehenswürdigkeiten, Bars, Geschäfte und mehr auf einer Route zwischen zwei Punkten vorzuschlagen. Benutzer können Start- und Endpunkt innerhalb einer Stadt angeben und zwischen einem Algorithmus ohne und einem mit zusätzlichen Einschränkungen wählen. Letzterer berücksichtigt auch noch Zeit- und Budget-Höchstgrenzen des Benutzers. Dann schlägt das System, basierend auf den Benutzerinteressen, zwei Routen zwischen den beiden Punkten vor: ein Baseline-Ansatz und eine Route, die unsere eigene Implementierung, bestehend aus mehreren Verbesserungen, darstellt. Insbesondere haben wir den Pearson-Korrelationskoeffizienten integriert, damit die vorgeschlagenen Plätze in der Route besser mit den Interessen der Benutzer korrelieren. Darüber hinaus haben wir sowohl die Klassifizierung, als auch die Zeit- und Budget-Schätzungen der einzelnen Orte verbessert. Auf der Website hatten die Benutzer die Möglichkeit, die zwei Pfade anhand mehrerer Kriterien zu bewerten. Die Auswertung des Feedbacks hat ergeben, dass unsere Implementierung auf signifikante Weise verbessert hat, wie gut die Plätze auf dem Pfad mit den Benutzerinteressen zusammenpassen. Insgesamt waren die Benutzer mit unseren eigenen Algorithmen zufriedener als mit denen der Baseline.

Contents

Chapter 1

Introduction

This introductory chapter will give an overview of the thesis. We will make motivation and problem statements, look at related work, and will give an overview of both the website project and this written report.

1.1 Motivation

Visiting an unfamiliar city can be exhausting. It is often stressful to find one's way around using public transport and navigate to the hotel. But it doesn't end there. Unless you already know exactly what you want to see in the city beforehand or you are with someone who knows the place, it can be quite hard to find the best venues to go to during the visit. Of course, every city has tourist attractions that are easy to spot, but these are not necessarily the most interesting or exciting places. It can take a long time to get personal tips for the trip that are not overcrowded with tourists. Also, it might be hard to organize the venues you want to visit in day trips in a way that most of the day isn't spent riding around between different places.

To make planning a city trip easier, a recommender system can be used. These applications try to suggest interesting, custom items, in our case places in the city, to the user. That way, users can more easily decide which places to visit and can make the most out of their limited time in the foreign city.

1.2 Problem Statement

In this thesis we implement a recommender system for city trip planning and investigate how to generate paths between two points within one city. There are multiple requirements for the system. First, it should be accessible through the Internet, i.e. in form of a website. Users should be able to enter start and end point and then get a route that consists of entertaining venues and is at the same time not too long. Also, we want the system to adjust the trip to the user's settings, which include their preferences for certain categories and optionally a time and budget limit for the trip. The computed route should be displayed on an interactive map on the website.

For evaluation purposes, we want to additionally display a baseline implementation on the map and let users rate the two routes based on different criteria. That way we can tell if our own implementation is actually an improvement over the baseline approach.

1.3 Related work

For the baseline approach we rely on the results of [Ilti 14]. In that Master's Thesis, three algorithms were developed: one exploration algorithm to retrieve and classify places, a constraint-free, and a constraint-based algorithm for path generation. A prototype website was set up to display the results of the algorithms. Then, a user study was conducted to compare the constraint-free and the constraint-based path finding algorithms against each other.

Our own implementation of the place exploration and the two path finding algorithms is at its core also based upon this implementation, but has some major differences and improvements regarding all three algorithms. Since the original code was written in Java, and we decided to use JavaScript on the website, we had to completely re-implement all of the algorithms.

1.4 Overview of the functionality of the website

The first thing the user sees when visiting the website is an instructional text explaining what they are supposed to do. They can rate six different categories on a scale from 0 to 5, where 0 means that no places from that category are going to be suggested and 5 means that this category will be strongly preferred, if possible. Next, they can optionally enter a time and budget limit for the trip which will be considered for the trip alongside the preferences. Then, after entering a start point and an end point in the two specified text fields, the two routes are computed and displayed on the map; though the user doesn't know which is the baseline and which is our own implementation, as the colors of the paths are distributed randomly. Using the two overview tables below the map, the user can get a good picture of the different paths. Even further below, there is a box with several questions about the routes that the user is supposed to answer. After clicking the submit button, the feedback is sent back to us.

1.5 Outline of this thesis

In the following chapters of this thesis we will thoroughly explain the process of creating the recommender system. First, we will present the technologies and APIs that were used for the site. Afterwards, a detailed explanation of the website design and the implementation of the place exploration algorithm, the two path finding algorithms, and the special algorithm to display the paths follows. Here, both the baseline approach and our own implementations and improvements will be described. Finally, we will elaborate on the user study and analyze the results to determine which of the two implementations is the better choice when planning a city trip.

Chapter 2

Used technologies and APIs

In this chapter we will discuss which technologies, frameworks, and APIs were used for the website and the reasons for our choices.

2.1 Google Maps API

To display the paths between start and end point, some sort of map needed to be shown on the website. There are several frameworks available that allow embedding a map, e.g. OpenStreetMap, Bing Maps, and Google Maps. For this project we decided to use the current web version of Google Maps, the “Google Maps JavaScript API v3”. This decision was made for a couple of reasons. For one, as Google Maps is so extremely popular, most users are already familiar with the user interface, which makes it more comfortable for them than having to get used to some other UI. Also, the API provides a lot of well-documented functionality, which made development easier and quicker. And of course, it works on all major platforms and browsers, including mobile.

2.1.1 Features

Google Maps supports all features that were important for our application. These will be discussed in the following sections.

2.1.1.1 Pricing

First, since this project is a non-commercial website, it was important that the framework is free to use. Fortunately, Google Maps offers a free version. And even though it comes with some minor limitations, these were not too severe for us (see section 2.1.2).

2.1.1.2 Markers and info windows

Being able to display markers on the map is a crucial feature of the maps framework. Without it, we could not mark the positions of the venues that we want to recommend, which is the main reason why we need this map in the first place. The fact that Google Maps supports markers in many different colors and shapes, even allows us to distinguish between different categories. But the functionality doesn't stop here: it is also possible to animate markers, make them draggable (which we use for the markers of start and end point), and to display custom info windows for them. We make these info windows appear when a user clicks on a marker to display additional information, such as the rating, price tier, and so on.

2.1.1.3 Walking path

In order to display a path that actually uses walkways to connect the places and not just draws straight lines between markers, the Google Directions API is used. Google has a huge database of walkways, which even contains paths that can not be accessed by car but solely by foot. This is an important feature for us, as the route is actually intended to be walked by foot. Using the Directions API, we can draw the walkable path through the recommended places on the map.

2.1.1.4 Geocoding service

The Geocoding API is another important service provided by the Google Maps API. It allows to convert addresses into coordinates and vice versa. We use it to derive a human-readable address from the latitude and longitude values we get from the Geolocation API (see section 2.2). That way we can quite accurately display the address of the position of the user in the start point text field if they agree to use their current position as start point. Similarly, if the user drags the start or end point marker, we can convert the new position (of which we initially only have the coordinates of) to an actual address. That is much nicer for the user to read than some coordinates.

2.1.2 Limitations

There are some limitations with the Google Maps API; some of which don't affect a research project of this scale much, but should be considered when expanding the project to a larger scale. Though for some limitations we had to find a workaround (see section 3.5.2).

2.1.2.1 Amount of allowed map loads and direction requests per day

The limits of the free version of Google Maps are generally very generous. It allows up to 25,000 map loads per day, which is more than enough for such a small-scale project. On top of that, this limit can be exceeded for 90 consecutive days before payment is enforced by Google. [Googb] A similar restriction applies to direction requests, of which 2,500 can be made per day. [Googa] Direction requests are made every time a path is displayed on the map, i.e. every time someone enters two valid places and these places are connected to form a walkable path. 2,500 requests should still be enough if only one request was made per route. However, for almost every route displayed on the map more than one request is necessary due to Google Maps also limiting the amount of waypoints on every path. This limitation will be described in the next section.

2.1.2.2 Amount of waypoints on a path

The Google Directions API allows to draw a path on a map that connects two points though a walkable route and even allows for waypoints to be set between start and end point. These waypoints are perfect for our application, for we don't have to make a new request for every adjacent pair of points. However, only up to eight waypoints are allowed per request. [Googa] Since most paths contain a lot more than eight venues, this is not sufficient for our application. We can work around this limitation by splitting the path in connected sub-paths, each of which consisting of the maximum amount of waypoints allowed, and making a separate request for each of them. Doing so, however, drastically increases the number of direction requests. Assuming that for every pair of routes on average 8 requests have to be made, only about 300 total requests can be made per day. For this project this should still enough, though we are getting rather close to the limit. The precise algorithm will be described in more detail in section 3.5.2.

2.1.2.3 Query limit per second

Splitting the paths into multiple sub-paths and making a separate request for each of them provokes another limitation of the Google Maps Directions API. By making so many requests in rapid succession, we quickly reach the query limit of the API because only ten requests are allowed to be made per second. [Googa] Like the other two limitations of the API, this constraint can be circumvented by using an appropriately designed algorithm to display the paths on the map. In the particular case of the query limit, we simply enforce a delay between two requests. Again, the exact functionality will be described in section 3.5.2.

2.2 W3C Geolocation API

Our application lets the user specify to use their current position as start point. This is particularly useful if the user is currently in the city and is accessing the website using their phone. In this real world scenario the user would not want to enter their current position manually, especially since practically every smart phone nowadays has GPS support. Because of that, we added a button next to the start point text field that uses this information and sets the user's current position as start point. To access the position, we use the W3C Geolocation API. It allows us to get the current latitude and longitude of the user based on a variety of different indicators, including GPS, the user's IP address, GSM cell IDs, and others. [W3C]

Because this information is very sensible, it requires the user's explicit consent before it can be accessed. That means that every browser that supports the API will show a pop-up window asking the user if they want to share their location with the website. If the user agrees, the coordinates are then retrieved and a callback function is called which, in our case, displays the location on the map and updates the corresponding start point text field. As of August 2015 more than 90% of all Internet users have a browser that supports the W3C Geolocation API. [Deve]

2.3 Foursquare API

It was decided to use the Foursquare API to find relevant venues in the area between start and end point. Another option would have been to use Google's Places API, which maybe would have seemed more natural since we already use the map by Google. However, prior comparison of the two APIs has revealed that the Foursquare API is indeed better suited for recommending touristic places. [Ilti 14, chapter 3.6] Also, it is no problem at all to display venues obtained from Foursquare on the Google Map as we can simply pass the latitude and longitude values directly from the Foursquare response to the map and display a marker there.

2.3.1 Authentication

In order to retrieve any data from Foursquare, authentication is inevitable.

2.3.1.1 Authentication method

Foursquare provides two methods for authentication: userless authentication and authentication using an access token via the OAuth protocol. The difference between the two is that the latter requires the user to log in with their Foursquare account and thus can provide more personalized data. [Foura] However, we decided to use userless authentication for two reasons. First, having to log in before being able to use a website is an additional hurdle which might deter many potential users, especially if they fear that any private data from their profile might be passed to a third party. Of course, we would not let this happen, though the uncertainty and wariness of the users remains. Second, many people don't use the Foursquare App itself which means that authenticated access wouldn't improve the results for the majority of users.

2.3.1.2 Client ID and secret

Making a userless request to the Foursquare API requires an API key. The key consists of ID and secret which are basically two strings used to make an HTTP request to retrieve information from Foursquare. [Foura] While the client ID can be made visible to the user, the client secret has to be kept private by all means. This is the main reason why the Foursquare request can't be made client-side but has to be made server-side, in our case using PHP. Basically, the client sends a request consisting of source- and destination-coordinates to the server which then performs the Foursquare request using the app ID and secret and sends the data back to the client. Then the browser can display this information.

The other reason why we run the Foursquare request server-side is that we can minimize the amount of data transferred to the client. Since we don't need all the fields that the Foursquare response contains (see section 2.3.2), we let our server put together the relevant pieces of information and remove the rest of it before sending the response back to the client. That way, the amount of data the client has to download upon making a request can be drastically reduced. The actual size of the data is only a fraction of the size of the original data received from Foursquare. For example, when using "Marienplatz Munich" as start point and "Munich Central Station" as end point, the size of the data from Foursquare adds up to 348 KB, whereas the data sent to the user is only 59 KB before compression. That is merely 17% of the original size. After compression the actual transmitted file size drops down to 12 KB.

2.3.2 Provided data

As described in the previous paragraph, the Foursquare API returns a lot of data which we don't need for our application, like formatted address, user comments, links to photos, and more. [Fourb] In this section we elaborate more on what pieces of information we actually extract from the received data for later use. First, we obviously need ID, name, latitude, and longitude of each place in order to show the place on the map. Also, category name and ID are important to correctly classify each place. For creating a place rating, we use the checkins, likes, rating, and rating count properties. Lastly, in order to give the user some additional information about the place, we also extract whether the place is open right now, how many people are there right now, and its price tier.

2.3.3 Usage limit

Just like the Google Maps API, the Foursquare API applies some usage restrictions. Fortunately these don't affect us as much as the Google Maps limitations do, meaning that we don't have to specifically adopt our algorithms to the limitations. Most importantly, 5,000 userless requests can be made per hour. [Foure] Even though we make multiple Foursquare-requests every time the routes are updated (see section 3.2.2.1), we didn't even reach 5,000 Foursquare requests during the entire evaluation period, let alone in one hour. Furthermore, unlike Google, Foursquare doesn't enforce a query limit per second, which means we don't have to manually slow down our requests. In short, the usage limits of the Foursquare API were no concern to us.

2.4 Front end: Bootstrap and jQuery

For the front end we use the relatively widespread frameworks Bootstrap 3.3.4 [Twita] and jQuery 1.11.2 [jQuea]. Additionally we use the Bootstrap Toggle plugin v2.2.0 [Hur] and the Bootstrap slider plugin version 4.8.3 [Kemp] for more UI elements.

2.4.1 Advantages

Using web frameworks to create a web page has several advantages over using pure HTML, CSS, and JavaScript.

2.4.1.1 Responsive design

Most importantly, Bootstrap was designed to make it easy to develop web apps that work on all devices, desktop and mobile. Nowadays, many users use their phone or tablet to browse the web, and, in many cases, use these devices more often than their desktop computers. Furthermore, since this recommender system is meant to be used on the way through a city, users would eventually have to use the website on their phone to view the recommendations. Bootstrap does a very good job in giving the developer the right tools to make the website responsive for all display sizes.

2.4.1.2 Modern, consistent design

Another advantage of Bootstrap is its modern design. Plain HTML buttons and other input elements look a bit out of place and old fashioned on a modern website and it takes extensive styling to make them look decent. Bootstrap's input elements, on the other hand, have a very modern, good-looking design. Also, there are lots of elements available which would take some time to recreate without Bootstrap, e.g. add-ons to the left and right side of text fields, button groups, tabs, modals, and the grid-layout, to name a few. Using all of these pre-designed input elements saves a lot of work and ensures a consistent design.

2.4.1.3 Built-in functions

Bootstrap requires jQuery to be included in the website. However, we don't just use jQuery because Bootstrap doesn't work without it. jQuery provides many useful functions that make it easier to manipulate HTML and CSS from JavaScript code. Especially the jQuery selector function `$(...)` is a powerful tool to easily access HTML elements. Also, jQuery's `$.ajax()` function makes it easy to reload resources from the server asynchronously. This is necessary in order to send the coordinates to the server and then receive the places from the Foursquare API.

2.4.2 Browser support

For a web application exposed to a broad audience it is important to make sure we support as many browsers as possible. However, for reasons of efficiency, we do not support outdated browsers, like Internet Explorer 8, whose market shares are negligible. All major browsers from the past five years should be able to correctly display the site, though. The browser support of Bootstrap and jQuery corresponds to this strategy. [Twitb] [jQueb]

2.5 Back end

For our back end we entirely rely on conventional, commonly used software. The underlying operating system of the web server is Ubuntu 12.04.2 LTS.

2.5.1 Apache HTTP Server

We decided to use the Apache HTTP Server 2.2.22 [Apac] because it is the most widely used web server software. Also, it is very easy to configure and set up. For example, since on our machine there was already another service running on port 80, we were able to let the Apache Server listen to port 8000 simply by changing the "Listen"-entry in the file `/etc/apache2/ports.conf`. No additional setup was required. Putting all HTML, CSS, and JavaScript files in the `/var/www/html/` directory sufficed.

2.5.2 PHP and PostgreSQL

A database was necessary in order to store the feedback that users submit on our website. Also, we needed a back end which handles both the Foursquare requests and access to the database. For this task, we decided to use PHP 5.3.10 [PHP c] in combination with PostgreSQL 9.1.16 [Post].

PHP is fairly widely spread and allows for relatively easy file- and database-access. Moreover, it is very easy to read parameters from an HTTP request and to send a response. This is important for our Ajax request which invokes the Foursquare query on the server and then returns the results to the client.

For our database we use the relational database management system PostgreSQL because it is open source and also fairly wide-spread. Furthermore, it implements the latest SQL standard. However, since we only have one table and only do insert operations, it doesn't matter too much which database management system is used, as long as it is relational. Setting up PostgreSQL was pretty straight-forward. After downloading the according packages, PostgreSQL required us to create a role using the `createuser` command and then setting its password through the Linux user "postgres" using the `psql` command line tool. Then the database had to be created using the `createdb` command. After a restart of the Apache server, everything was set up.

Chapter 3

Design and Implementation

This chapter gives a detailed explanation of how the algorithms for place exploration and path finding work. Also, we will elaborate on the algorithm which displays the results on the map. The majority of code is written in JavaScript and runs client side; the rest of the code is written in PHP.

3.1 General overview

First, we will give an overview of all the JavaScript and PHP files and illustrate the call sequence. This will help understanding the more in-depth explanations of the individual program parts later on.

3.1.1 JavaScript files

The JavaScript code is divided into seven files with about 2,500 lines of code.

The script `ui.js` manages the user interface. It is responsible for dynamically adjusting the UI elements on the page, including sliders, switches, tables, and the feedback options. When the website is first loaded, `ui.js` uses jQuery's `$(document).ready()` function to initialize all elements on the page. The script's most important function is `updateUI()` which will be described in section 3.1.3.

After `ui.js`, `map.js` is the script with the most lines of code. It encompasses everything necessary to correctly display the map on the website. That includes the map itself, markers, info windows, paths, but also the Google Maps Autocomplete input forms and their event listeners. When the user enters two places, these event listeners are the ones to trigger the `updateUI()` function in `ui.js`. The path rendering algorithm in this file will be discussed later this chapter (see section 3.5.2).

The source files `classification_baseline.js`, `classification_ownImpl.js`, `algorithms_baseline.js`, and `algorithms_ownImpl.js` contain the actual algorithms for classification of places and for path generation (i.e. the constraint-free and the constraint-based algorithms). We use a separate namespace for each of the two implementations so that we can easily distinguish between them. These scripts will be described in sections 3.2.3, 3.3, and 3.4.

Finally, `utilities.js` contains several helper functions for sorted arrays, calculating distances based on coordinates, and Pearson's correlation coefficient (see section 3.3.2.2). Additionally, it defines constants which are used by the other scripts, e.g. the full human-readable name for each of our six predefined categories.

3.1.2 PHP files

Besides `index.php`, which mostly contains markup, there are four other PHP scripts with about 800 lines of code in total.

The file `foursquareQuery.php` makes the Foursquare API call, selects the information relevant for our application, and sends it back to the user's browser. For more information, see sections 2.3.2 and 3.2.1.1.

For database access, we use the scripts `feedback.php` and `feedbackAdministration.php`. The former is called via Ajax by `ui.js` when the user submits his feedback and adds a new data record to the database. The other script is only used internally to display the database table and show some statistics. Also, it contains code to reset the database. Of course, all database access is performed using SQL.

The last PHP script, `createCategoryIndex.php`, serves a special purpose. It is never called by the end user, but only run internally to create the `categories.json` file which is used to accurately classify Foursquare venues. See section 3.2.3.1 for more details on this.

3.1.3 Main call sequence

Figure 3.1 illustrates the call sequence for the `updateUI()` function. This function is called after the user has selected their preferences, has chosen the algorithm that should be used (constraint-free or constraint-based), and has entered two places. Then this function checks if these places are no more than five kilometers apart from each other. This restriction is enforced because the algorithms are designed for city trips that can be walked by foot and would not give as good results for longer distances. That is because for trips between cities, the distance heavily outweighs the entertainment-factor, which are both used to determine the venues for the route (see section 3.3). Consequently, the algorithms would only return routes that don't differ much from the linear distance between start and end point. Besides the distance of the places, if the user has selected the constraint-based algorithm, `updateUI()` also verifies that the values in the text fields for time and budget constraints are indeed numbers. Moreover, the time limit must not exceed 500 minutes and the budget limit must not be higher than 1000€ in order to prevent nonsensical inputs.

After checking all user input, `updateUI()` makes the Ajax call to the server for place exploration, invokes the algorithms for classification and path generation, and finally renders the results on screen. In short, the sequence diagram in figure 3.1 basically represents the main program logic of the application.

For simplicity reasons, this diagram does not differentiate between baseline approach and our own implementation. In reality, separate ***`categorization`*** and ***`algorithm`*** calls to each implementation's scripts are made. Also, the ***`categorization`*** actually consists of multiple functions (see section 3.2.1.2) which are condensed into one call here. And of course, depending on the user's selection, the ***`algorithm`*** call will be one of the two available algorithms, i.e. either `constraintFreeAlgorithm()` or `constraintBasedAlgorithm()`. The function calls `addPaths()` and `addPlaceTables()` are used to display the results on the website and will be discussed in sections 3.5.2 and 3.5.3 respectively.

3.2 Place exploration and classification

The process of generating a path from start to end point can be split up into two subtasks. First, potential candidate venues have to be determined and rated, and then another algorithm can find the most suiting route consisting of a subset of these places. The first

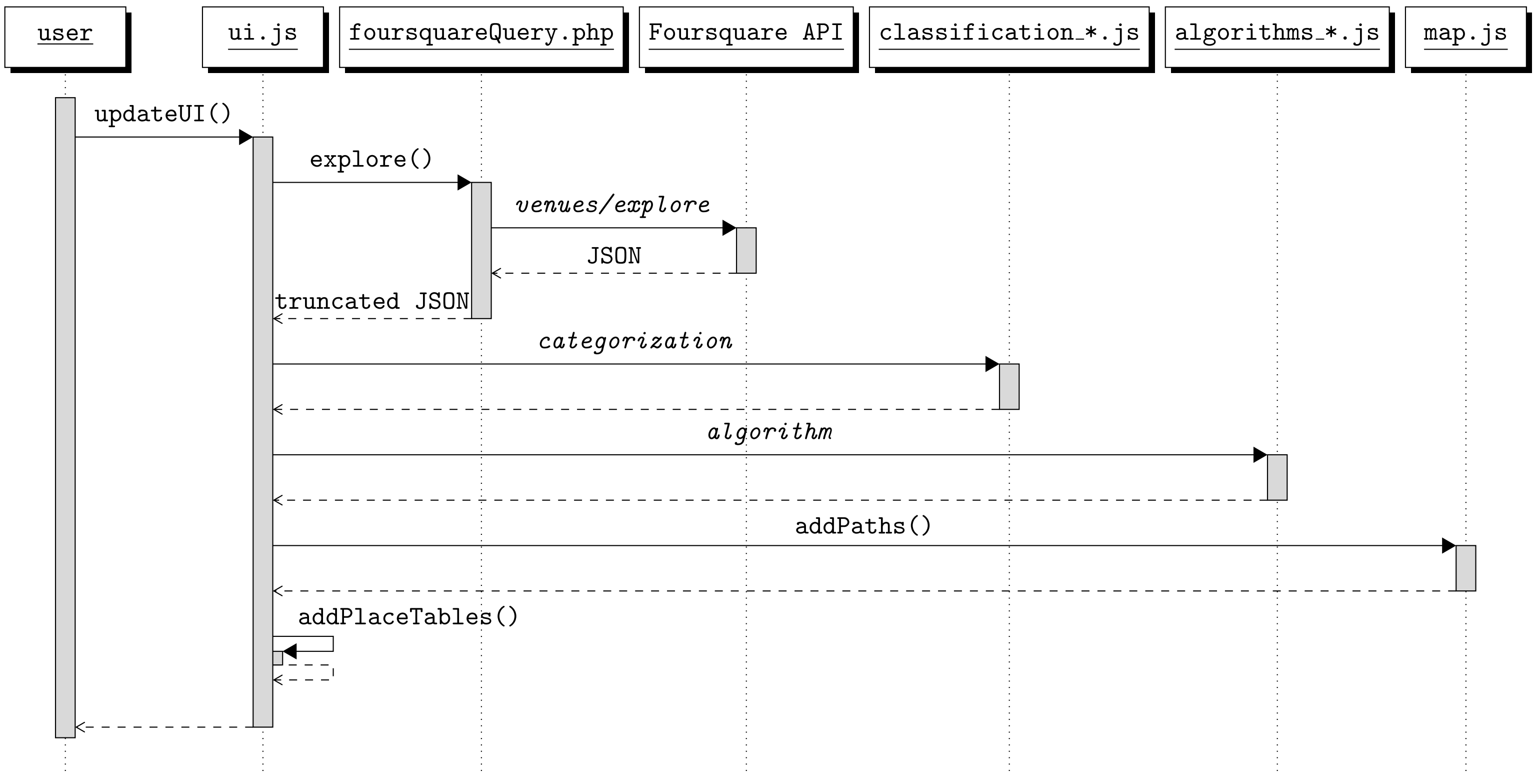


Figure 3.1: Call sequence diagram

part is called place exploration and will be discussed in this section. To start with, we will give an overview of of both the baseline approach and our own implementation; afterwards, our own implementation will be discussed more in-depth.

3.2.1 General description

Some parts of the exploration algorithm have to run server side for reasons explained in section 2.3.1.2. However, the majority of the algorithm runs client side to save server resources. The following sections explain the basic structure of server side and client side code.

3.2.1.1 Server side code

When the user enters two valid places, an Ajax request is made which basically calls the script `foursquareQuery.php` with source and destination coordinates as parameters. The script then calculates the midpoint and the distance of these two points. Using these values and the hard-coded Foursquare client ID and secret, our server makes a Foursquare

API call to the `venues/explore` endpoint. In section 3.2.2 we will go into more detail about what queries are actually made. After retrieving the JSON files from Foursquare, we decode them, read the desired information (see chapter 2.3.2) and write it in a new array. This array is then again encoded as JSON and finally sent back to the client using PHP's standard output.

3.2.1.2 Client side code

After the server has made the Foursquare query, on the client side a callback function in the script `ui.js` is called which first parses the received data as JSON object, and then passes it to the `classification_baseline.js` and `classification_ownImpl.js` scripts for further processing. However, since the original baseline approach only takes the first 100 results of the Foursquare query into account and no places gotten by additional queries, two different JSON objects are passed to the respective scripts. This is intentional behavior, as some of our improvements over the baseline approach involve adjusting the server side part of the exploration algorithm as well. For more information on the extra places that our implementation considers, see section 3.2.2.

The baseline script implements the exploration algorithm of [Ilti 14, chapter 3]. That approach works in three steps: first, the places are categorized based on a hard-coded list of predefined categories. Then, for each place three scores are calculated: checkin score, likes score, and rating score. For that, the categories are treated separately. Basically, if a place has a higher value of the according property, it gets a higher score than another place in the same category with lower property value. Now, for each place the total score is calculated as the average of the three scores. Finally, that score is multiplied by the user rating of the place's category. Because the original algorithm was written in Java, we rewrote it in JavaScript ourselves based on the Java code and the Master's thesis so that it can be integrated in our website.

For our own implementation we took a different approach which is influenced by the baseline but differs in some major aspects. Most importantly, the categorization of places is no longer based on a hard-coded list, but done dynamically (see section 3.2.3). Also, the scaling has been adjusted (see section 3.2.4) because most of it is done in the actual algorithm now (see section 3.3.2). Another important aspect of our exploration algorithm will be explained in the next section.

3.2.2 Maximizing the amount of places for the exploration function

A great effort was made to receive as many places as possible from Foursquare. This is crucial for the classification and path finding algorithms because if there is a larger input set, naturally they can pick the best places for the route. We were able to achieve a higher number of places than the baseline through two different ways.

3.2.2.1 Making multiple Foursquare queries

Since the usage limitations of the Foursquare API are very liberal (see section 2.3.3), we were able to perform multiple requests for each invocation of the exploration algorithm. Especially in city centers where there are very many places in high density, Foursquare finds hundreds of interesting places. However, when querying the `venues/explore` endpoint it will return at most 100 places, even if the `limit` parameter is set to a higher number. If we want more places, we need to make an additional request and set the offset parameter to 100 in order to get the next 100 results. As the places returned by Foursquare are ranked by relevance, these venues may be less relevant generally, but might still be interesting for the user if they rank a specific category very highly. Additionally, we perform one more request to the `venues/explore` endpoint in which we include the `&section=sights` parameter. This option turned out to return many interesting places that the normal request doesn't contain. There are many more options that could be set, including food, drinks, arts, outdoors, and more. [Fourb] We decided to only include sights because for this category the normal Foursquare query without `section` parameter returned only very few venues, and the new query added a lot of new places to it.

In that regard, we also took caution not to do too many requests so that the waiting time for the user does not become too high. It was important to find a good balance between amount of places and loading time, because these two values are inversely proportional to each other. Depending on the network it already takes several seconds to get the reply from the server. That is why we only perform up to two general queries, even if there are more places (which is only very rarely the case) and then do one query with the `&section=sights` parameter. All in all we send a maximum of three queries to Foursquare for each invocation of the exploration algorithm.

3.2.2.2 Adjusting the radius

Another way to get as many relevant places as possible is to increase the search radius. The baseline approach calls the Foursquare API with the center between start and end point and a radius of half of the distance between the two. However, it turned out that if the radius is set to 1.2 times the actual radius, more relevant places at the start and end points appear in the final route.

3.2.3 Classification algorithm

Next, we will give an in-depth explanation of our classification algorithm. The algorithm's task is to assign each place one of our six predefined categories that are rated by the user beforehand ("Sights & Museums", "Night Life", "Food", "Outdoors & Recreation", "Music & Events", and "Shopping") and filter places that can easily be detected as inappropriate.

3.2.3.1 Eliminating unclassified places

The Foursquare API already tells us the category of each venue, however, these categories are very specific and must be condensed into fewer categories which the user can rate individually. In our case, we use six predefined categories that provide a very good coverage of places. The baseline approach used a hard-coded and incomplete list of categories to sort places based on their Foursquare category, but since there are more than 700 categories [Fourc], and that list is continuously updated, it naturally missed a lot of places and wrongly discards these unclassifiable venues. In some cases, it even assigns the wrong categroy to a place. That is why we use a dynamic approach that does not accidentally miss any categories, for it uses the complete list of categories. This is possible, because the Foursquare categories are already arranged in a hierarchy that defines top level categories that are similar to ours. Table 3.1 shows which Foursquare top level categories correspond to the categories used in our project. One minor adjustments had to be made, however: Foursquare's "Arts & Entertainment" category corresponds mostly to our "Music & Events" category, but its "Museum" subcategory should of course show up in our "Sights & Museums" category. We use a blacklist to sort these venues correctly.

Unfortunately, the API does not explicitly tell us the parent category for each subcategory. Using the category tree from the `venues/categories` endpoint, we can only determine the child categories. That means we have to create a list ourselves that stores the top

Own category	Foursquare top level category
Sights & Museums	Professional & Other Places
Night Life	Nightlife Spot
Food	Food
Outdoors & Recreation	Outdoors & recreation
Music & Events	Arts & Entertainment
	Event
Shopping	Shop & Service
Other	College & Education
	Residence
	Travel & Transport

Table 3.1: Comparison of categories

level category for each subcategory available so that the classification algorithm can later quickly assign one of our six categories to each place based on its Foursquare category.

For that purpose we created the `createCategoryIndex.php` script which downloads all the categories currently available from Foursquare as a JSON object. Then for every top level category it recursively goes down the category tree to find all subcategories. For every subcategory the script writes a new attribute inside one large JSON object. The attribute key is the subcategory ID and the attribute value is an object containing the subcategory name and the ID and name of the corresponding top level category. This JSON object is then stored in the file `categories.json` (see figure 3.2).

After creating this index once, every time we have to classify Foursquare places we can use the index to quickly look up the top level category for every category available and won't accidentally misclassify one. We simply load the JSON file using jQuery's `$.getJSON()` function upon loading the website and store it in the global variable `categoriesJSON`. Then for every category ID we can call `categoriesJSON[`*`id`*`].parentCatID` to get the top level category. Depending on that value, we assign the corresponding category of our own to that place.

Because Foursquare continuously updates their category tree, the `createCategoryIndex.php` script has to be run regularly in order not to miss any new categories. In fact, only two weeks after the public launch of the project, a new food category, "Trattoria/Osteria", was added, and the script had to be rerun to find it. However, running the script every time a new Foursquare request is made is not advisable, as it would slow down the application and cause a lot of unnecessary traffic.

```
{
    "4d4b7104d754a06370d81259":{
        "parentCatID":"4d4b7104d754a06370d81259",
        "parentCatName":"Arts & Entertainment",
        "catName":"Arts & Entertainment"
    },
    "4fceea171983d5d06c3e9823":{
        "parentCatID":"4d4b7104d754a06370d81259",
        "parentCatName":"Arts & Entertainment",
        "catName":"Aquarium"
    },
    ...
    "4bf58dd8d48988d10d941735":{
        "parentCatID":"4d4b7105d754a06374d81259",
        "parentCatName":"Food",
        "catName":"German Restaurant"
    },
    ...
}
```

Figure 3.2: Excerpt from `categories.json`

3.2.3.2 Deleting inappropriate places

Since this application is intended to be used by tourists, we decided to remove some places that tourists would rather not want to visit. Based on observations of reoccurring places, we created a list of inappropriate categories: “Fast Food Restaurant”, “Multiplex”, “Supermarket”, “Bakery”, “Drugstore”, “Pharmacy”, “Movie Theater”, “Phone Shop”, “Furniture”, “Grocery”. The idea is that tourists would very rarely want to visit these kinds of venues. Of course, there might be exceptions for every case on the list, but for the most part these places are not desirable, and therefore removed from our dataset.

3.2.3.3 Deleting places with bad rating

After categorization, our exploration algorithm removes places that don’t have a Foursquare rating, or whose rating is too bad. If a place doesn’t have a rating, it means, that not enough Foursquare users have checked in or left reviews for it yet. Because these places seem to be not very popular, we can a priori remove them and thereby reduce the size of our dataset. A smaller amount of total nodes means that the actual path finding algorithm will later run faster.

Furthermore, if any of the six categories has more than ten venues, we also remove some of the worst-rated places from that category. We use the following formula to determine the amount of places to remove, depending on the user rating for that category:

$$amountOfPlacesToDelete = 0.40 - 0.05 \cdot userPreference \tag{3.1}$$

So for a category with more than ten places and a user rating of five, we remove the worst 15%. If that rating is one, we remove 35%. Note that if the rating is zero, no place of that category will show up in the final route, as we completely filter them out later.

3.2.4 Scaling of places

The baseline approach implements a very simple strategy to let user preferences determine how often a place appears in the actual route by simply multiplying the place rating of each venue by the user rating of the according category. That way, places with lower rating would still likely appear in the final route if the user preference for that category is high.

Our own implementation, however, uses a different way to make sure places with high preference get preferred. We basically leave the preference scaling up to the path finding algorithm and only remove places in categories with a user preference of zero in the exploration phase. The exact algorithm of the user preference scaling is explained in detail in section 3.3.2.

Also, unlike the baseline, we do not use checkins and likes values from the Foursquare API, but solely rely on the total Foursquare rating and the amount of votes (i.e. how many people rated the venue). We don't need to put together our own rating by incorporating as many sources of information as possible when the Foursquare rating already includes all of this data. [Fourd] Therefore, we use the rating as is and only scale it by how many people rated the place, meaning that if two places have the same rating, the place with more votes gets a higher score. Since the amount of votes is not as important as the actual rating itself and since the amount of votes can be quite high, a logarithmic scale is applied:

$$totalScore = rating \cdot \log_2(amountOfVotes + 1) \tag{3.2}$$

We add one to the amount of votes so that if a place has no votes, it doesn't evoke a mathematical error. That score is the entertainment value used by the path finding algorithms later on.

3.3 Constraint-free algorithm

After all the places in the area have been categorized and ranked, the path finding algorithm creates the final route from a subset of these places. Two algorithms, a constraint-free and a constraint-based one, were implemented for this task. We will start with the constraint-free algorithm, which just tries to find a short and entertaining path from start to end point without taking any more constraints into account. To begin, we will give a broad overview of both the baseline and our own implementation and after that we will elaborate on our own approach in more detail.

3.3.1 General description

Once again, we use Iltifat's approach as a baseline for the constraint-free algorithm, which is described in detail in [Ilti 14, chapter 4.2]. The algorithm basically generates a weighted, complete graph from all places found by the exploration algorithm where the edge weight is defined as the physical distance between its two vertices. Then Dijkstra's algorithm is used to find the best path from start to end point. However, we don't search for the shortest path, as in a compete graph that is obviously always the direct route from start to end. Instead, we maximize the fraction $\frac{entertainment}{distance}$ for each subpath. Here, *entertainment* is the sum of the scores of all venues on the path with the individual scores being the ones previously determined by the exploration algorithm. And *distance* is the total path length. So when the algorithm compares two paths from start point to a certain node, it prefers the one for which that fraction is biggest. The rest of the algorithm is exactly the same as for the standard Dijkstra's algorithm. Since the original code is written in Java, we also rewrote that algorithm in JavaScript.

Our own implementation takes this baseline approach as basis, but additionally incorporates a correlation coefficient in order to make the amount of places for each category in the route correlate to the user preferences better. This will be described in the following sections.

3.3.2 Correlation of user preferences and venues

Most of the time, the amount of places in each of our six predefined categories returned by the exploration algorithm varies greatly among the different categories. Especially in the food category there are often much more venues than in any other category. The result is that a path calculated with the baseline algorithm will often contain many restaurants, even if the food category is set to a lower preference. Meanwhile, other categories that the user rated highly are neglected because there are only few places available which makes it more unlikely for them to be taken into account by the algorithm. That is why we looked for a way to make sure the actual amount of places in each category correlates with the user preference.

3.3.2.1 General idea

The idea is that the amount of places of each category in the final path should not be dependent on the total amount of places in that category found by the exploration algorithm. It should rather be dependent on the user rating for that category. If one category has a rating twice as high as another category, then in the path there should be roughly twice as many places of the first category than there are places for the second category. Though, notice that by looking at the rating of a certain category alone, it cannot be determined how many places there should be in this category. Only by looking at the ratings of all categories, we can say which categories should have more and which should have less places as part of the final path. For example, if the user rates all categories the same, then there should be about the same amount of places in each category; however, we can not say what that amount should should be exactly. It depends on the context. If start and end point are far away from each other and the exploration algorithm finds a lot of places for every category, we could suggest maybe ten places for each category. In contrast, if start and end point are close together and there are not as many places in the near vicinity, maybe only one or two places can be suggested for each category. The only thing that we can say for sure is that all categories should be represented in the final path to the same degree. Again, we should proceed like this even if one category, e.g. the food category, has disproportionately many relevant venues.

3.3.2.2 Pearson's correlation coefficient

To implement this idea, we use the most widely used correlation coefficient, namely Pearson's correlation coefficient. Given two datasets $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ of size n, it lets us calculate a measure of the linear correlation of them. [Weisa]

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{3.3}$$

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{3.4}$$

$$r(x,y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \cdot \sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{3.5}$$

Pearson's coefficient r (equation 3.5) gives values between -1 (indicating perfect negative correlation) and $+1$ (perfect correlation), with 0 meaning no correlation exists between the datasets.

In our case, the two datasets x and y are user preferences for the six categories and the amount of categories in the current subpath since we want these two properties to correlate as closely as possible. For a description how exactly this coefficient is incorporated in our algorithm, see section 3.3.3.

3.3.2.3 Sample Variance

For our application there are two special cases when Pearson's coefficient does not give a meaningful value, and thus have to be treated separately. The first one is pretty straightforward: if both samples are constant, then the denominator of the Pearson coefficient would be zero. However, since both samples are the same except for a constant factor, we can return 1 in this case, indicating perfect correlation.

The second special case occurs if only one of the datasets is constant. This is an important scenario, because users can quite likely set all their preferences to the same value. In that case, the denominator of the Pearson coefficient would also be zero, and we can't meaningfully set the coefficient to a fixed value. We use sample variance in this situation, which is a measure for how far off the dataset is from the sample mean. [Weisb]

$$\mathrm{Var} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{3.6}$$

Unlike Pearson's coefficient, though, sample variance is not limited. In order to be able to use these values in the same context, we need to scale the variance to values between 0 and 1, where, just like above, 0 means no correlation, and 1 means perfect correlation. This is only possible because the input values are limited between 0 and 5. Therefore, we can calculate the maximum variance, which is $\frac{1}{2} \cdot \left(\left(0 - \frac{0+5}{2}\right)^2 + \left(5 - \frac{0+5}{2}\right)^2\right) = 6.25$. Now, because the minimum variance is 0, we can scale the sample variance with $1 - \left(\frac{\mathrm{Var}}{6.25}\right)$.

3.3.3 Balance between correlation coefficient, rating of places and walking distance

The baseline approach maximizes the fraction $\frac{entertainment}{distance}$ for each subpath. For our own implementation, we multiply that fraction by our correlation coefficient because we want the value to be as big as possible. Further tests showed that the algorithm yields better results if *entertainment* and *distance* values are weighted more than the correlation coefficient, which is why we square them. This means that our algorithm tries to find the path from start to end point which maximizes the following value:

$$\frac{r(preferences, amountOfPlacesPerCategoryInPathSoFar) \cdot entertainment^2}{distance^2} \tag{3.7}$$

Again, *entertainment* is the accumulated rating score of all venues on the path which we receive from the exploration algorithm and *distance* is the total path length. So when comparing two paths from start point to a certain node, our algorithm prefers the one for which that fraction is biggest.

3.4 Constraint-based algorithm

An alternative way to create a path from a subset of the explored places is implemented with a constraint-based algorithm. Unlike the algorithm described in the previous section, this algorithm now takes time and budget limits for the route into account which the user can enter on the website. Next, we will give a broad overview of both the baseline and our own implementation and will then talk about our own approach in more detail in the following sections.

3.4.1 General description

For the baseline against which our own algorithm is evaluated, we again use Iltifat's algorithm as described in [Ilti 14, chapter 4.3]. In order for it to fit into our project, we had to re-implement it in JavaScript, just like the exploration and the constraint-free algorithm. The algorithm works similar to the constraint-free algorithm with some additional elements. At its core, it is still based on Dijkstra's algorithm to find the best path between start and end point. However, as an additional preliminary step, each venue, depending on its category, is assigned a fixed value for cost and time to spend there. Then, a weighted complete graph is created using the physical direct distance between nodes as edge weight. Using this information, before a subpath is compared against another path in Dijkstra's algorithm, it is checked whether the following conditions are fulfilled:

- subpath does neither exceed time nor budget limit
- subpath does not contain more than one restaurant or more than one venue from the night life category

If one of these conditions is not met, the subpath will be rejected; otherwise, if another valid subpath from start point to this current node has been found prior to the current path, the two subpaths are compared against each other using the *entertainment* value of each path. Just as before, *entertainment* is the sum of the scores of all venues on the path with the individual scores being the ones previously determined by the exploration algorithm. The subpath that has a bigger *entertainment* value will be saved together with the node. Note that in accordance with [Ilti 14] this part of the algorithm differs from the constraint-free algorithm in that this time we don't include the physical distance between nodes for the comparison of subpaths.

We take this implementation as a basis for further improvements. We focus on better heuristics for time and budget estimations for each venue and incorporate Pearson's correlation coefficient. Our changes will be described in the following sections.

3.4.2 Estimation of time and cost for each place

Since the baseline approach uses fixed, hard coded values to estimate how expensive a venue in a certain category is and how much time users want to spend there, this was the area where the most improvements could be made. However, it is not a trivial task to assign these time and budget values to each place, for a couple of reasons. Obviously, every place is different, and while the broad categorization into six categories divides the places into groups of similar venues, places within each category can still differ greatly in their cost and time requirements. Also, these values are highly dependent on the individual user. Some users might want to spend more time at a museum than others, and some people spend more money in a restaurant than others. All in all, it is impossible to make estimations that fit everyone's needs. That is why our estimations are only guidelines that help getting a general idea of the place requirements.

3.4.2.1 Time heuristics

For activity duration, prior research has already derived heuristics for some of our six categories based on a real-world Foursquare data set with 3.7 million users and 300 million checkins. [Meli 12] Especially the estimations for how long users spend at restaurants were useful to us. The data set revealed that on average, users spend 53 minutes eating lunch, slightly less for breakfast and a bit more for dinner. These values seem fairly accurate, and so we use 45 minutes as a rough base estimation of the time to spend at all venues in the "Food" category. Depending on the user's preference for the "Food" category, this value will be adjusted later (see section 3.4.2.2).

However, the other values derived from that data set didn't fit our application particularly well either because the categories that were used in the study didn't match our own six categories (like "Work" or "College & Education"), or because the time values for venues in the categories were much too long for a city trip. For example, the average time for both "Arts & Entertainment" and "Parks & Outdoors" derived from the data set is about five hours each. The same applies to the "Shops" category, where the average activity duration is supposedly a little under four hours. These time limits are totally inappropriate for our application because most people would like to do multiple activities on a city trip and thus not spend so much time at one location. Also, users would rarely enter a time limit that is longer than a couple of hours. So in order to provide interesting routes, we don't use these time limits but instead use more realistic estimations as shown in table 3.2.

Category	Time estimation
Sights & Museums	60 minutes
Night Life	45 minutes
Food	45 minutes
Outdoors & Recreation	30 minutes
Music & Events	50 minutes
Shopping	40 minutes
Other	60 minutes

Table 3.2: Time estimations for each category

These values are chosen with respect to typical venues in each of the six categories. None of the values are above 60 minutes in order to support routes with multiple attractions, even if the user enters a relatively low time limit. However, these limits are not fixed and in the next section we will describe how they are adjusted depending on the user's preference for the specific category.

3.4.2.2 Using user preferences to determine time constraints for categories

The only personal data we have available from the user are their preferences. Consequently, the best way to estimate a user's willingness to spend time at a certain venue, is by using their specific preferences for the venue's category. If a user rates a category highly, then they will likely spend more time there. In the previous section we already defined a base value for the time consumption of each category. This basis will now be adjusted based on the user's rating of the category. Table 3.3 shows for each possible user rating by how much the time to spend at a venue is offset from the base value. Of course, if a category has rating 0, then places in that category won't show up in the final path at all and an offset is pointless.

User rating	Time offset
1	–15 minutes
2	–5 minutes
3	0 minutes
4	+5 minutes
5	+15 minutes

Table 3.3: Time offset depending on the user's rating

The notion that the higher rated a category is, the more time a user wants to spend there is pretty logical. However, there is one downside of this idea. If we add too much time for venues whose categories are rated highly, places from these categories will end up occurring much more rarely in the final route because adding these places to the route makes it more likely that the user's time limit is exceeded. On the other hand, places from categories with a lower rating would come up more often because adding another one of these places would increase the total time not by much. So in order to make good time estimations for the user and at the same time prevent the algorithm to prefer low-rated categories, we kept the offset values at a maximum of 15 minutes difference.

3.4.2.3 Foursquare's cost heuristic

Estimating how much money a user will spend at a certain venue is easier than time estimation because Foursquare already sorts places into four price categories as shown in table 3.4.

Upon closer investigation, this categorization seems fairly accurate and will be used for our algorithm. Unfortunately, not every Foursquare venue has a price tier assigned. This can either be due to not enough people having checked in there yet, or because a fixed price

ID	Description	Price range	Converted price
1	Cheap	$< \$10$	8€
2	Moderate	\$10 – \$20	16€
3	Expensive	\$20 – \$30	24€
4	Very Expensive	$> \$30$	32€

Table 3.4: Foursquare price tiers

assignment would not make sense for that place. That is especially true for venues in the categories “Outdoors & Recreation” and “Shopping”. For the most part, only restaurants and bars are classified this way by Foursquare.

So if a place, for any reason, does not have a price tier assigned, we fall back to a hard coded assignment based on the place’s category. Table 3.5 shows the fallback values for each category.

Category	Default cost estimation fallback
Sights & Museums	8€
Night Life	15€
Food	15€
Outdoors & Recreation	5€
Music & Events	15€
Shopping	10€
Other	10€

Table 3.5: Cost estimations for each category

3.4.3 Incorporating Pearson’s correlation coefficient

In addition to making money and time estimations more accurate, we also improved the constraint-based algorithm itself. For the constraint-free algorithm we implemented Pearson’s correlation coefficient in order to make the places in the route correlate better with the user’s preferences (see section 3.3.2). The same idea can be applied to the constraint-based algorithm. However, since the baseline approach only compares the *entertainment*-factor of each subpath and does not include the physical distance when comparing subpaths, we do the same for our own implementation. Therefore, the value which the algorithm maximizes is the following:

$$r(preferences, amountOfPlacesPerCategoryInPathSoFar) \cdot entertainment \tag{3.8}$$

So when the constraint-based algorithm finds two subpaths from the start point to a certain node, which both fulfill the user's constraints, the path that is chosen is the one which maximizes this value.

3.5 Displaying the results

After the routes have been computed, they need to be displayed on the website. For a good user experience we not only show the paths of both implementations on the map, but also display a summary of all the places in either path in the form of two tables. Also, we need to put up the feedback options so the user can rate the two routes. In the following sections we will elaborate on how the website was designed in general and how the results are displayed to the user.

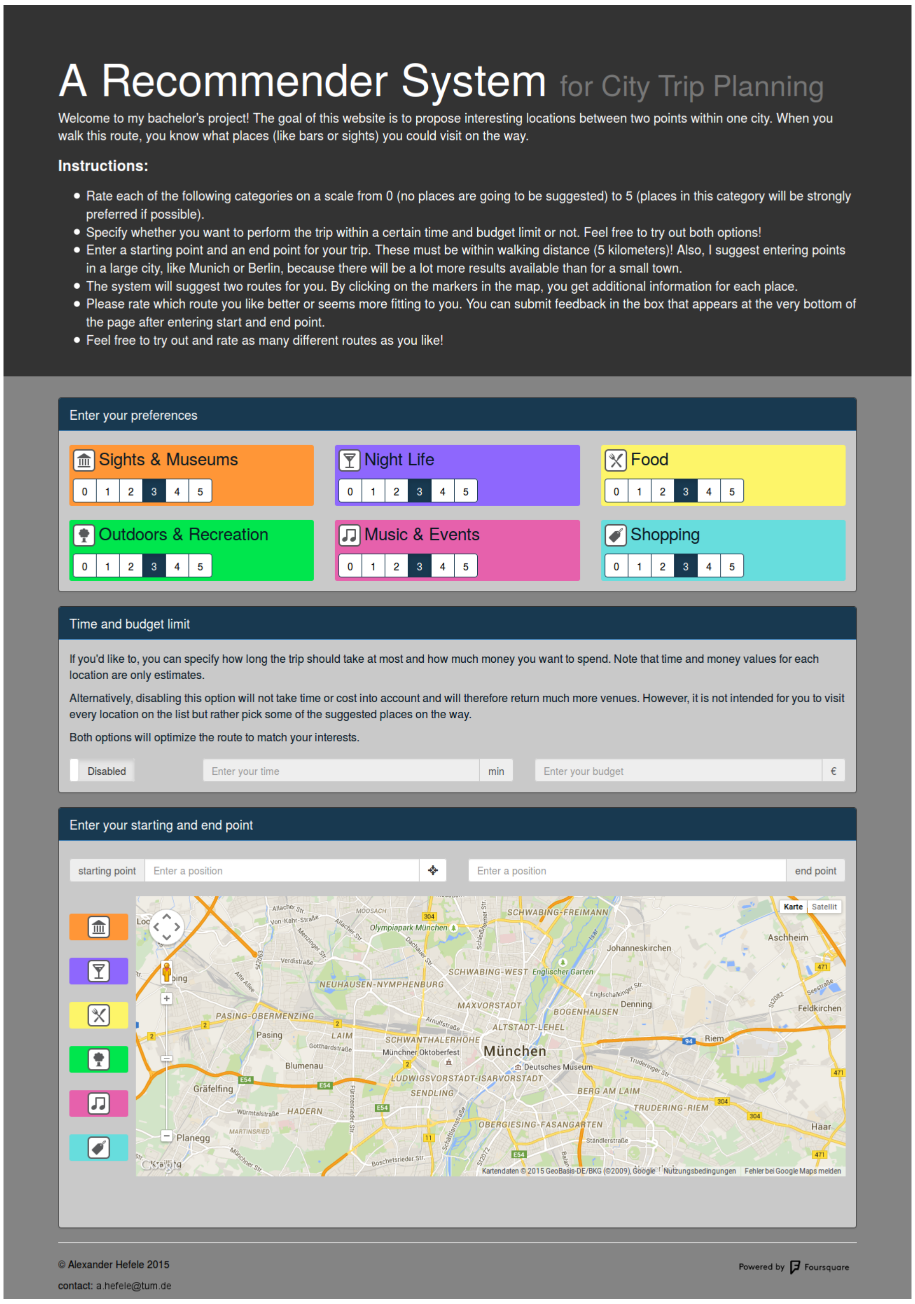
3.5.1 Single-page design

A single-page application (SPA) is a website that only consists of one HTML file. We decided to use a single-page design for our website because not having to click through multiple pages and instead seeing everything in one spot makes it very easy to use. By dynamically loading content and displaying it using JavaScript, we can respond to user input despite not having any actual links.

The page itself is structured as follows. At the top of the page, there is a short introductory paragraph about what the site is about and some instructions on how to use the application. Below, there is a container where the user can enter their preferences for all of the predefined categories using six button groups. Each category has a different color and a distinct icon. Underneath, another box tells the user more about the two path generation algorithms. There is a switch to toggle between them and next to that are two text fields for time and budget limits. Next, there is a box with another two text fields and a map. In the text fields the user can enter start and end point of their path, which will immediately be displayed in the map. Left to the map, there are the same icons and colors for each of the six categories as above so that the user can quickly see which colored marker in the map corresponds to which category.

When the user has entered two valid positions and the paths have been calculated, additional UI elements appear on the page. First of all, the two routes from start to end point are drawn on the map, one being blue and one being red (for more detail see section 3.5.2). Right underneath the map, two switches are added which can be used to hide individual paths for better clarity in the map. Also, labels for walking distance and time for each of the paths show up right there. Below that, two tables appear which give an overview of all the places in either path. At the very bottom of the page a box with several feedback options is made visible where the user can rate the paths and send their opinion.

Figure 3.3 shows what the website looks like before any user input has been made.



A Recommender System for City Trip Planning

Welcome to my bachelor's project! The goal of this website is to propose interesting locations between two points within one city. When you walk this route, you know what places (like bars or sights) you could visit on the way.

Instructions:

- Rate each of the following categories on a scale from 0 (no places are going to be suggested) to 5 (places in this category will be strongly preferred if possible).
- Specify whether you want to perform the trip within a certain time and budget limit or not. Feel free to try out both options!
- Enter a starting point and an end point for your trip. These must be within walking distance (5 kilometers)! Also, I suggest entering points in a large city, like Munich or Berlin, because there will be a lot more results available than for a small town.
- The system will suggest two routes for you. By clicking on the markers in the map, you get additional information for each place.
- Please rate which route you like better or seems more fitting to you. You can submit feedback in the box that appears at the very bottom of the page after entering start and end point.
- Feel free to try out and rate as many different routes as you like!

Enter your preferences

Category	Rating
Sights & Museums	0 1 2 [✓ 3] 4 5
Night Life	0 1 2 [✓ 3] 4 5
Food	0 1 2 [✓ 3] 4 5
Outdoors & Recreation	0 1 2 [✓ 3] 4 5
Music & Events	0 1 2 [✓ 3] 4 5
Shopping	0 1 2 [✓ 3] 4 5

Time and budget limit

If you'd like to, you can specify how long the trip should take at most and how much money you want to spend. Note that time and money values for each location are only estimates.

Alternatively, disabling this option will not take time or cost into account and will therefore return much more venues. However, it is not intended for you to visit every location on the list but rather pick some of the suggested places on the way.

Both options will optimize the route to match your interests.

Disabled | Enter your time min | Enter your budget €

Enter your starting and end point

starting point | Enter a position | Enter a position | end point

Karte
Satellit
München
Aschheim
Johanneskirchen
Feldkirchen
Riem
Haar
Pasing
Blumenau
Gräfelfing
Planegg
Kartendaten ©2015 GeoBasis-DE/BKG (©2009), Google Nutzungsbedingungen Fehler bei Google Maps melden

© Alexander Hefele 2015
contact: a.hefele@tum.de

Powered by Foursquare

Figure 3.3: Screenshot of the website

3.5.2 Displaying the paths

When implementing an algorithm to draw the paths on the map, the three limitations described in section 2.1.2 had to be taken into account since we do not just display straight lines between the places in the route, but draw an actual walking path that can be walked by foot and have to use the Google Directions Service to do so. This section will explain how exactly we circumvented the restrictions of the API and were able to display the paths on the screen nevertheless.

3.5.2.1 Path rendering algorithm

After the routes for both the baseline approach and our own implementation have been computed, the function `addPaths()` in `map.js` is called. First, for every place in either route, this function calls `addPlaceMarker()` which adds a new marker and an info window to the map using the Google Maps API. The info window can be made visible by clicking on the marker and displays additional information: name of the venue, category, Foursquare category, checkin and like count, how many people there are right now, rating, in which route this place is in (blue, red, or both), and, if available, price tier and whether the place is open at the moment.

Now that the markers are visible, we still need to display the path between them. Before we can do this, however, some preliminary steps are necessary, which are performed by `prepareRouteRequests()`. This function sets up the `directionDisplayRequests` objects and initializes the Google Maps `DirectionsRenderers`. The former are later used to make a request to the Google Maps Directions Service and retrieve the path information which is then used by the `DirectionsRenderers` to finally draw that path on the screen. Multiple `directionDisplayRequests` and `DirectionsRenderers` have to be defined because each one of them can only handle one path consisting of a start point, and end point, and a maximum of eight waypoints due to limitations of the Google Maps API (see section 2.1.2.2). Hence, `prepareRouteRequests()` splits both of the two routes (baseline and our own implementation) into multiple smaller connected paths that fit this criterion. Every subpath can consist of up to ten points (start point, end point, and eight waypoints in between). However, in order to display a connected path, the start point of one path must be the same as the end point of the previous path. So that means if a route has ten or less venues, one path is needed; for eleven to 19 venues, two paths are needed; 20 to 28 places require three paths, and so on. Generally, if a route has n places, the total amount of subpaths necessary is $\lceil \frac{n-1}{9} \rceil$. In `prepareRouteRequests()` the routes are split according to this formula and start point, end point, and the waypoints of each subpath are written into a new `directionDisplayRequests` object which will later be passed to the Google Directions Service to request this specific subpath.

Once the routes are split into paths that can be handled by the Google Maps API, `renderAllRoutes()` is called. That function initiates a chain of API calls to the Google Maps Directions Service. Every API call sends a `directionDisplayRequests` object consisting of up to ten locations to the Google server and returns an object which can be used by a `DirectionsRenderer` to draw the path on the map. So using the very first `directionDisplayRequests` object, `renderAllRoutes()` makes the initial API call for the first subpath of the first route.

When the path has been computed by Google, a callback function is called with a *response* and a *status* parameter. Our callback function in turn adds an *attempt* and *implementationID* parameter in order to keep track of the different subpaths which we split the route into, and passes it together with the received data to `processRoute()`. That function's task is to use the response to display the according subpath on the map and to initiate another API call for the next subpath. Figure 3.4 shows a flowchart of this function to accompany the following explanation. At first, the status parameter is checked. If the status is `OK`, then the `setDirections()` function of the next unused `DirectionsRenderer` is called with the *response* parameter from the Directions Service API call, which essentially displays the path on the map. Next, the global index variable which determines the next unused `DirectionsRenderer` is increased so that the next subpath can display their subpath as well. Every `DirectionsRenderer` can only display one subpath at the same time. Then, we determine the walking time and the distance for this subpath which we accumulate in two global variables. To do that, we have to go through all the so-called legs of the subpath. A leg is a part of the subpath that connects exactly two consecutive locations. So a subpath with start point, end point, and eight waypoints consists of nine legs. We simply go through all the legs of the subpath and add their `duration` and `display` values to the according global accumulators. Once we have gone through all the legs of the subpath, the next step is to check if every `DirectionsRenderer` previously defined for this route has been assigned a subpath now. If that is not the case, we know that there are still some subpaths to render and thus make another API call to the Directions Service with the callback function set as `processRoute()` again. In case we have gone through all subpaths of the current route, we display the accumulated `walkingTime` and `distance` for this route below the map. Now, if at this point we have displayed both routes on the map (baseline and our own implementation), the function finally terminates. If, however, we have only displayed the first of the two routes, we make yet another API call with the first subpath of the second route, again with `processRoute()` set as the callback function.

In order to differentiate between the two routes, we use the *implementationID* parameter of the `processRoute()` function.

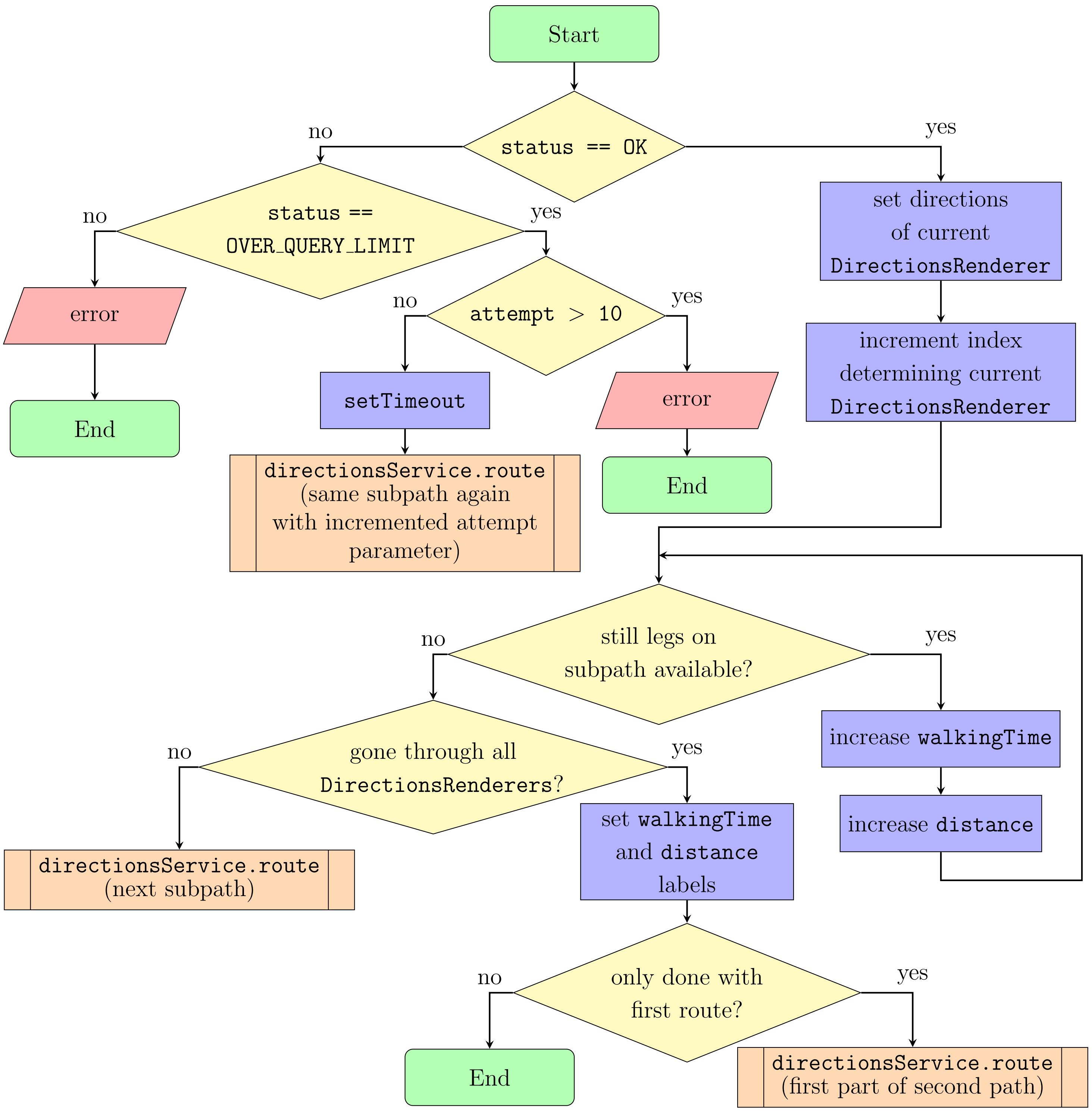

Start
status == OK
no
yes
status == OVER_QUERY_LIMIT
set directions of current DirectionsRenderer
no
yes
error
attempt > 10
no
yes
increment index determining current DirectionsRenderer
End
setTimeout
error
directionsService.route (same subpath again with incremented attempt parameter)
End
still legs on subpath available?
no
yes
gone through all DirectionsRenderers?
no
yes
increase walkingTime
increase distance
directionsService.route (next subpath)
set walkingTime and distance labels
only done with first route?
no
yes
End
directionsService.route (first part of second path)

Figure 3.4: `processRoute()` flow chart

A special case occurs if the status parameter returned by the Directions Service is not `OK`. Though, we can still cope with that situation if the status is `OVER_QUERY_LIMIT`, meaning that too many requests have been made in rapid succession (see section 2.1.2.3). In that case, we simply wait 500 milliseconds using JavaScript's `setTimeout()` function and then repeat the API call. We use 500ms because it seems to work every single time and isn't that much of a delay to become a nuisance for the user. After 500ms we should be below the limit of ten queries per second again. However, if, for some reason, that API call still exceeds the query limit after the delay, we wait again and repeat this procedure. To make sure that the algorithm does not make an endless amount of API calls, we abort after 10 unsuccessful attempts and display an error message. To keep track of the number of attempts, we use the `attempt` parameter of our `processRoute()` function which we increment when an unsuccessful attempt has been made.

If the status is not `OK` and not `OVER_QUERY_LIMIT`, then an error message for this unknown status is shown on screen and the function terminates.

3.5.2.2 Path color assignment

In order to make the evaluation as unbiased as possible, the color assignment of the two routes is random. The colors themselves, however, are fixed: one route is always red and the other one is blue. So if a user enters a new start point, end point, or both, the colors of the two routes are switched with a probability of 50%. We do not switch colors if the user only changes their preferences because that might cause confusion to the user.

We took great caution to make the color assignment as unnoticeable as possible. That is, all the other UI elements stay uninfluenced by the color switching. The table for the blue route is always on the left, and the one for the red route always on the right side of the screen. Also, the feedback options always show the blue route first, regardless whether it is the baseline approach or our own implementation. On top of that, we even always render the blue route first, and then the red route to give absolutely no hint on which is which. Only when submitting the feedback, we, invisible to the user, make sure to mark all feedback with the implementation ID of the corresponding route so that we can put it in our database correctly.

3.5.3 Displaying the tables

To give the user an overview of all the places in the paths, we display two summary tables; one for each of the two routes. These tables are added to the website below the map after the routes have been calculated. The function `addPlaceTables()` in `map.js` is responsible for creating these tables from the places retrieved from the algorithms. If the user has selected the constraint-free algorithm, the tables only consist of the name and category of each place; for the constraint-based algorithm, we additionally display the estimated time and cost for each place.

3.5.4 Showing feedback options

Lastly, upon entering two valid places, we also display the feedback options for our evaluation on the page. These options consist of five questions for each of the two implementations, one general question, a text field for additional comments, and a submit button. The exact method of feedback collection will be discussed in more detail in chapter 4.1. When the user submits their feedback, these UI elements will disappear and only reappear when another route request is made.

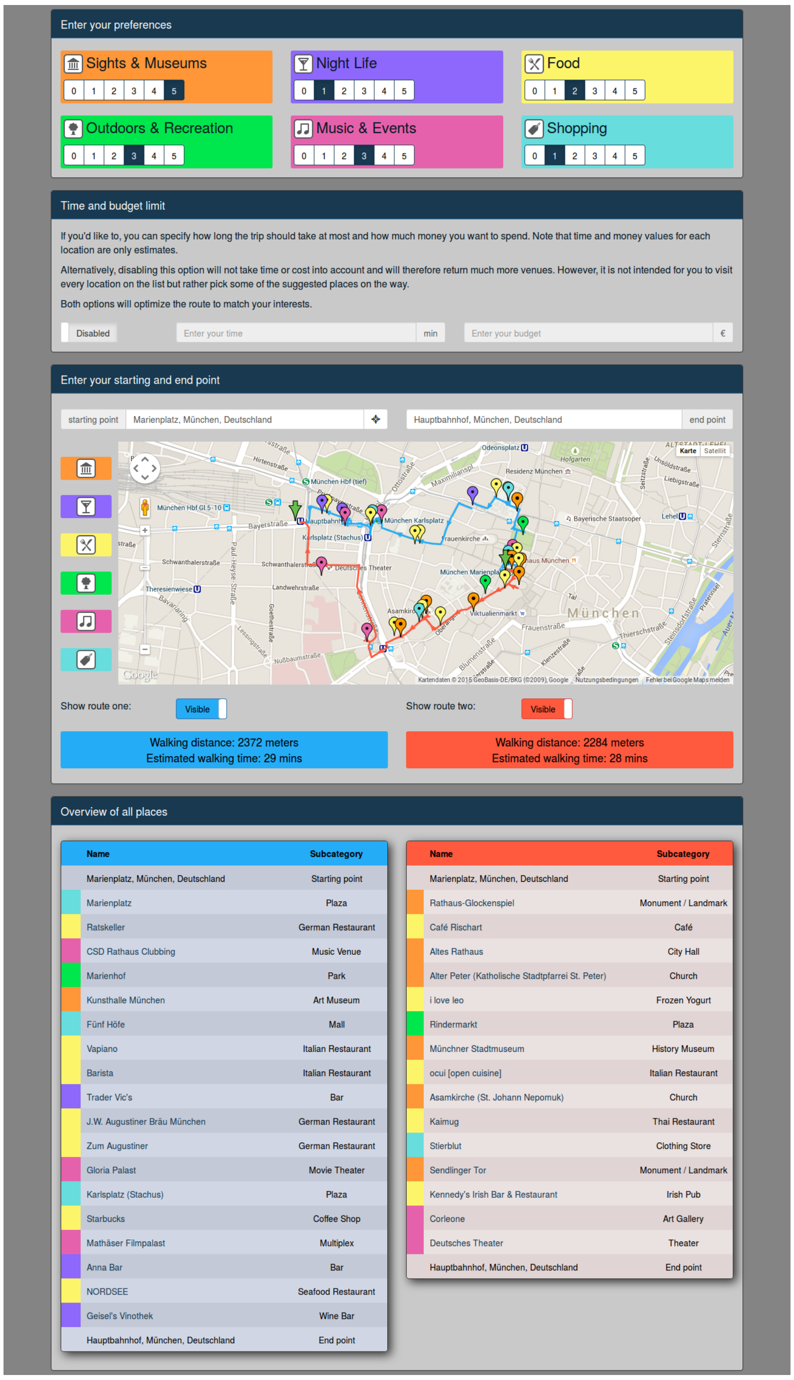
3.6 Illustrative example

The screenshot in figure 3.5 exemplary shows the differences between the two implementations that were described in this chapter. We entered Marienplatz in Munich as start point and Munich Central Station as end point and used the constraint-free algorithm. Also, note the particularly high preference selection of the "Sights & Museums" category. In the screenshot, the blue route is the baseline approach and the red route is our own implementation. Table 3.6 gives an overview of the selected preference and the amount of places in each category of both routes.

	Sights & Museums	**Night Life**	**Food**	**Outdoors & Recreation**	**Music & Events**	**Shopping**
User preference	5	1	2	3	3	1
Amount of places in baseline approach	1	3	7	1	3	3
Amount of places in own implementation	6	0	5	1	2	1

Table 3.6: Comparison table for the example Marienplatz – Munich Central Station

It can be seen that the baseline approach shows much less places from the "Sights & Museums" category despite it being rated so highly. Pearson's correlation coefficient for that route is -0.48. Whereas with six venues from that category, our own implementation follows the user's preference in all categories significantly better (Pearson's coefficient: +0.68). Our algorithm even makes a detour via "Sendlinger Tor" to lead the user to more sights while the baseline approach takes more or less the direct route.


Enter your preferences

Sights & Museums: 0 1 2 3 4 [✓ 5]
Night Life: 0 [✓ 1] 2 3 4 5
Food: 0 1 [✓ 2] 3 4 5
Outdoors & Recreation: 0 1 2 [✓ 3] 4 5
Music & Events: 0 1 2 [✓ 3] 4 5
Shopping: 0 [✓ 1] 2 3 4 5

Time and budget limit

If you'd like to, you can specify how long the trip should take at most and how much money you want to spend. Note that time and money values for each location are only estimates.

Alternatively, disabling this option will not take time or cost into account and will therefore return much more venues. However, it is not intended for you to visit every location on the list but rather pick some of the suggested places on the way.

Both options will optimize the route to match your interests.

Disabled | Enter your time min | Enter your budget €

Enter your starting and end point

starting point: Marienplatz, München, Deutschland
end point: Hauptbahnhof, München, Deutschland

Show route one: Visible

Show route two: Visible

Walking distance: 2372 meters
Estimated walking time: 29 mins

Walking distance: 2284 meters
Estimated walking time: 28 mins

Overview of all places

	Name	Subcategory
	Marienplatz, München, Deutschland	Starting point
🟦	Marienplatz	Plaza
🟨	Ratskeller	German Restaurant
🟪	CSD Rathaus Clubbing	Music Venue
🟩	Marienhof	Park
🟧	Kunsthalle München	Art Museum
🟦	Fünf Höfe	Mall
🟨	Vapiano	Italian Restaurant
🟨	Barista	Italian Restaurant
🟪	Trader Vic's	Bar
🟨	J.W. Augustiner Bräu München	German Restaurant
🟨	Zum Augustiner	German Restaurant
🟪	Gloria Palast	Movie Theater
🟦	Karlsplatz (Stachus)	Plaza
🟨	Starbucks	Coffee Shop
🟪	Mathäser Filmpalast	Multiplex
🟪	Anna Bar	Bar
🟨	NORDSEE	Seafood Restaurant
🟪	Geisel's Vinothek	Wine Bar
	Hauptbahnhof, München, Deutschland	End point

	Name	Subcategory
	Marienplatz, München, Deutschland	Starting point
🟧	Rathaus-Glockenspiel	Monument / Landmark
🟨	Café Rischart	Café
🟧	Altes Rathaus	City Hall
🟧	Alter Peter (Katholische Stadtpfarrei St. Peter)	Church
🟨	i love leo	Frozen Yogurt
🟩	Rindermarkt	Plaza
🟧	Münchner Stadtmuseum	History Museum
🟨	ocui [open cuisine]	Italian Restaurant
🟧	Asamkirche (St. Johann Nepomuk)	Church
🟨	Kaimug	Thai Restaurant
🟦	Stierblut	Clothing Store
🟧	Sendlinger Tor	Monument / Landmark
🟨	Kennedy's Irish Bar & Restaurant	Irish Pub
🟪	Corleone	Art Gallery
🟪	Deutsches Theater	Theater
	Hauptbahnhof, München, Deutschland	End point

Figure 3.5: Screenshot Marienplatz – Munich Central Station

Chapter 4

Evaluation

To determine if our own implementation is indeed an improvement over the baseline approach, we conducted a user evaluation. Over the course of a couple of weeks we had participants try out the website and rate the two routes based on multiple criteria. This chapter describes how exactly the feedback was gathered and discusses the results.

4.1 Feedback gathering

As mentioned in the previous chapter, we completely rely on a web-based submission form for collecting feedback from users. The advantage of this system over a paper survey is that we don't have to ask people in person and can reach a much wider audience by simply distributing a link to our website.

4.1.1 Feedback submission procedure

When the user hits the submit button, the function `sendFeedback()` in `ui.js` is called which sends all all necessary information to the server using an Ajax call to `feedback.php`. For more detail about what data is sent here, see section 4.1.3. Then the PHP script checks that all the parameters are set in the request and opens a connection to the database. Using the SQL command `INSERT INTO` it adds the new data record to the database and finally returns the success status to the browser. When `ui.js` receives this response, it hides the feedback options so that the user doesn't submit feedback for the same route twice.

4.1.2 Questionnaire

For our evaluation we use five question that are rated with a Likert scale. Table 4.1 gives an overview of the questions used in the feedback form and the provided answering options. Participants have to fill out these five questions for both routes so that a comparison between the two is possible. Analysis of the answers to these questions will follow in section 4.2.2.

ID	Question	Response options				
1	The total amount of places was...	too low	low	perfect	high	too high
2	The length of the path was...	too short	short	perfect	long	too long
3	How well did your received places match your preferences?	not at all	rather not	fairly well	quite well	perfectly
4	Would you consider taking this route yourself?	no		maybe	yes	
5	How satisfied are you with the overall result?	not satisfied	rather not satisfied	rather satisfied	quite satisfied	very satisfied

Table 4.1: Questionnaire for the evaluation

Additionally, there is one question asking the user directly which route they like better (evaluation in section 4.3). Also, we put a text field for additional comments, which was used fairly often by the participants (see section 4.2.4).

4.1.3 Collected data

We do not only send the user's rating to the server to be saved in the database, but also some information about the inputted data in order to make the submission more comprehensible. That encompasses submission time, start and end point, the user's preferences for all of the six categories, and which algorithm was used. If the constraint-based algorithm was chosen, we also save its constraints.

To get even more insight in the routes, we also save how many places there are on each route, how many places the respective explore function has returned, how long each of the routes is (in meters), and how long the direct distance between start and end point is.

Regarding user behavior, we record whether the user used the Geolocation Service (i.e. if the current position was used for that particular submission), how many routes the user has requested, and how often a user has submitted feedback so far. However, since we explicitly chose not to use any tracking mechanism in form of cookies or the like, these last two values are reset every time the page is refreshed and thus only contain limited information about the user's actual behavior.

The last value that is submitted when giving feedback at first might seem like a strange thing to save in the database, but has turned out to be a pretty wise addition once actual feedback submissions were made. We save the decision of the random generator whether the baseline approach or our own implementation is colored blue in the map. That information is not used to check if the random generator works properly, but rather to know which route a user means, if they use the comment box to refer to one of the colors, which actually happened multiple times during the survey.

We do not save any user specific information, like IP addresses or user agents in the database to make the survey as anonymous as possible. Also, no cookies or similar tracking mechanisms are used on the site. For our survey, an assignment of a dataset to a specific user is simply not necessary.

4.1.4 Security measures

Every web application that accepts user input is exposed to attacks. We took several security measures to prevent our website from getting compromised.

The biggest attack for web applications are injection attacks, specifically SQL injections. [Open 13] Without going into too much detail, using this technique, an attacker can basically execute arbitrary SQL code on the server by sending manipulated data using a text field or some similar input method on a website. Since we are accepting user input in form of the feedback submission options, we need to protect against this attack. Fortunately, with parameterized database queries SQL injections can be averted relatively easily. Therefore, we use the PHP function `pg_query_params()` instead of `pg_query()` for making the database query. [PHP b]

Another attack that we need to protect against, is Cross-Site Scripting (XSS). Because we display our results on a web page, an attacker could simply write HTML code containing JavaScript into the comment box, which would then be directly executed on our overview page. To prevent this, we don't display the user comments as they are submitted, but use PHP's `htmlspecialchars()` function which replaces the special HTML characters ‘&’, ‘”, ‘”’, ‘<’, and ‘>’ with their escaped equivalents ‘&amp;’, ‘&quot;’, ‘&#039;’, ‘&lt;’, and ‘&gt;’. [PHP a] As HTML code always requires tags covered in ‘<’ and ‘>’ signs, no foreign HTML and JavaScript can be executed on our site.

4.1.5 Finding participants

Getting enough participants for the study essentially meant distributing the link to as many people as possible. We used multiple sources to find people willing to take part in the evaluation. Especially Facebook turned out to be a very good way to reach thousands of people. We posted the link to several Facebook groups varying between 500 and 11,000 members. The idea is that if as little as one percent of the members of the biggest group visit the site, we still get more than a hundred page hits. Another good source of getting participants was to approach people personally. Many people only submitted feedback after individually asking them to do so.

4.2 Analysis of feedback

In this section we will finally get to the results of our user study. We will give some general statistics about the survey, evaluate the user ratings and analyze the other data that was collected.

4.2.1 General statistics

Because the whole survey can be done online, we were able to get much more participants than if we would have only asked people personally. During the evaluation period the website has been accessed over 600 times. Some of these visitors are recurring, but it is not possible to determine the exact number of unique visitors as we decided not to use cookies to track users. All in all, 533 routes have been computed, and 123 times did people actually submit feedback. The entire analysis in this section is based on these 123 data records.

With 85 submissions the constraint-free algorithm was much more popular than the constraint-based version which only 38 participants selected. There was, however, one functionality on the website that was even more unpopular: the Geolocation Service. Of the 123 submissions, only three used their current position. Now, this can be due to a couple of reasons. It could be that some people using the website live in a smaller town and wanted to try the application in a larger city, which we explicitly recommend in the instructions, and could therefore not use their current position. Or maybe some people did try out the location service but didn't want to submit feedback using it. For the outcome of the survey, however, this does not play too much of a role.

Other interesting statistics concern the distance of the routes. On average, the direct distance of the submitted start and end points was about 2,500 meters (with a standard deviation of σ = 1,200), which makes sense since the maximum allowed distance is set to 5,000 meters. The routes themselves were longer, of course. Surprisingly, the average for both routes was almost identical, namely 5,600 meters. The standard deviation is 3,000 meters for the baseline and 3,400 meters for our own implementation.

On average, the baseline route consisted of 14.3 venues (σ = 10.4), whereas our own implementation had 15.4 places (σ = 14.4), which is likely a consequence of the fact that, on average, the explore algorithm for the baseline only found 91.0 places per request (σ = 20.9), and our own implementation found 158.1 (σ = 63.3). That was to be expected since the latter is designed to make multiple Foursquare queries for each route request and thus returns more venues (see section 3.2.2.1).

Users have submitted feedback for city trips in cities all over the world, including Los Angeles, Amsterdam, Bilbao, Rome, London, Barcelona, New York, Paris, Turin, Hong Kong, Dublin, Porto, and many cities from Germany. The city with the most data records is by far, unsurprisingly, Munich with 46 submissions. Other popular cities are Berlin (11 submissions), Leipzig (7 submissions), and Hamburg (5 submissions).

Figure 4.1 the distribution of the submissions over time. On day five we posted the website in a fairly big Facebook group explaining the high peak. 98% of the submissions have been made within one week after initially sharing the website.

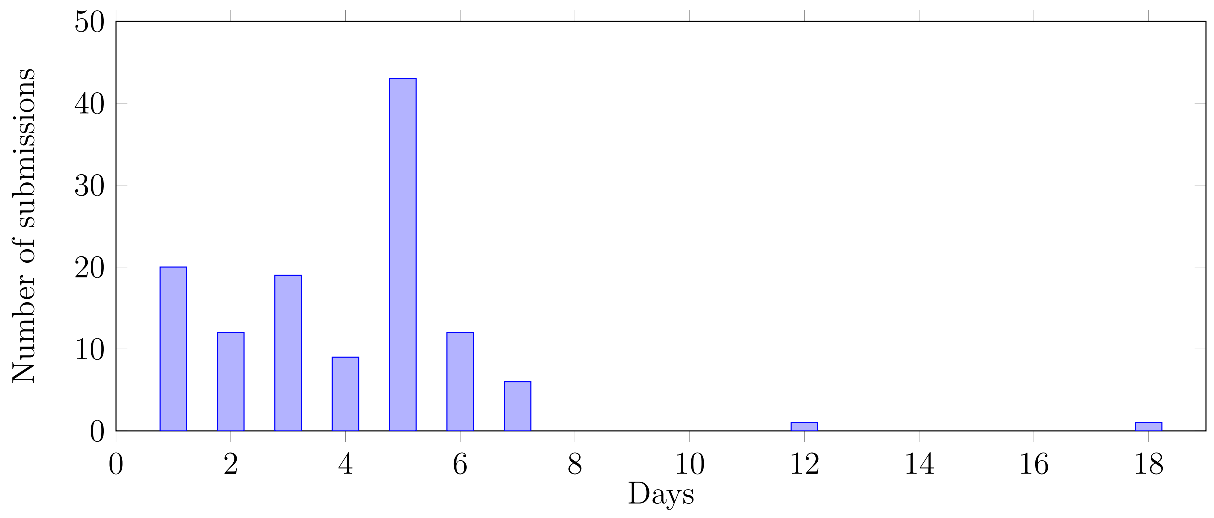


Figure 4.1: Feedback submissions over time

4.2.2 Evaluation of user ratings

Now we will analyze the actual feedback that was submitted by users, i.e. the rating of the five questions for each of the implementation.

Figure 4.2 shows the results of the first question which evaluated the amount of places in the routes. In this regard, the two implementations do not differ very greatly from each other. Generally speaking, many participants were satisfied with the amount of places, but there is still room for improvement. Another outcome of this graph is that everyone has other expectations for the route. Some people would have liked more places, some fewer places. It might be hard to achieve improvements here that fit everyone. Compared to the baseline approach, there hasn't been a change, neither for the better, nor for the worse.

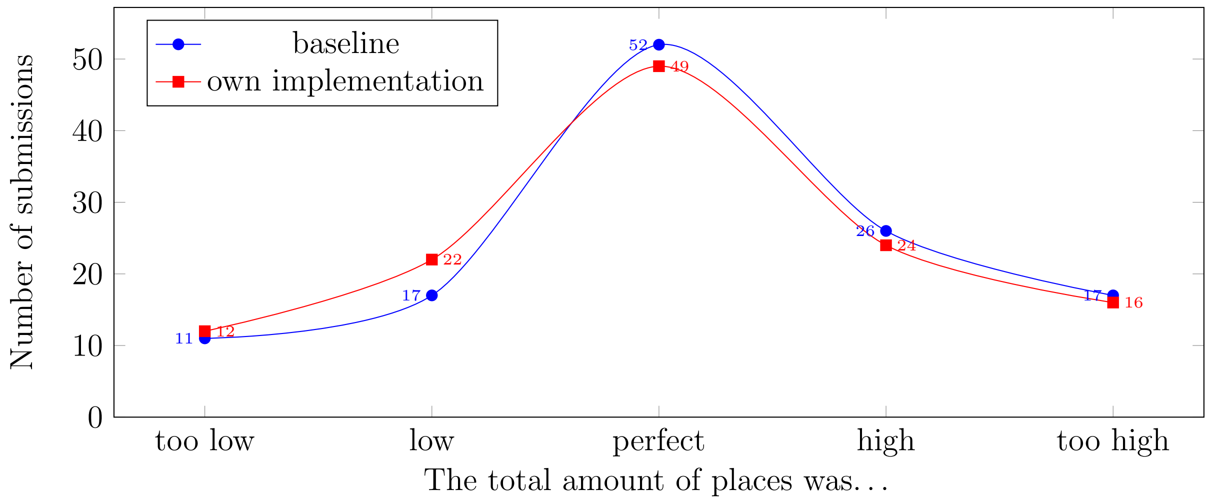


Figure 4.2: Results of the first question

In figure 4.3, one can see the user rating for the length of the paths. Just like for the first question, no significant difference between the two implementations can be detected. However, this bell-shaped curve is much steeper towards the middle than the previous curve, meaning that the variance from the perfect value and the standard deviation are much smaller. More than 70% thought that the length of the routes was perfect, but only about 40% were totally content with the amount of places. All in all, the length of the paths is pretty adequate.

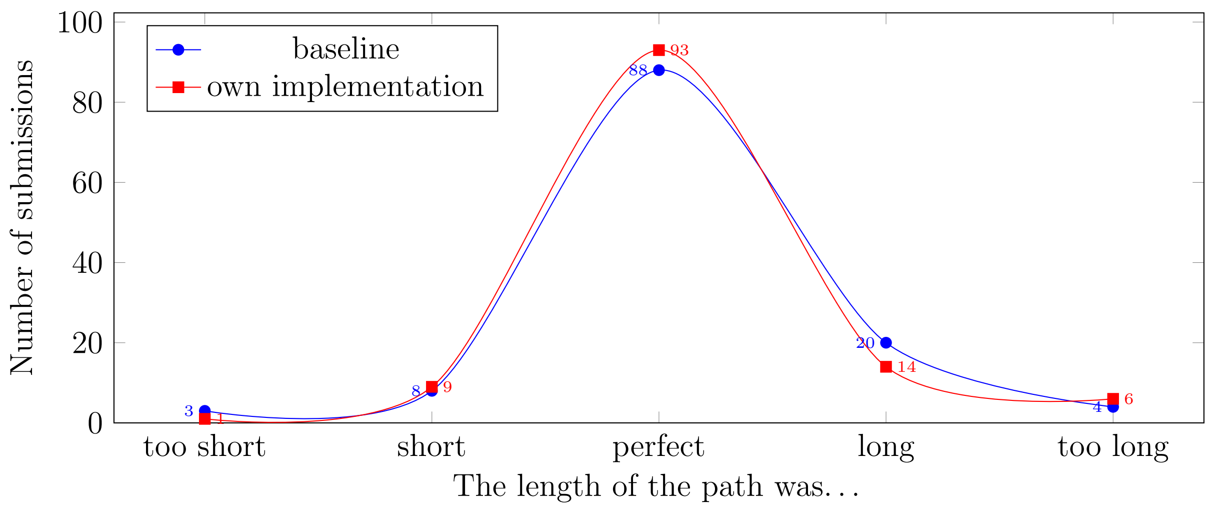


Figure 4.3: Results of the second question

The first real difference between the ratings for the two implementations can be seen in figure 4.4. We asked the users how well the places in the paths matched their preferences. For the baseline, about the same amount of people answered with “rather not”, “quite well” and “fairly well”. These three answers represent 80% of all responses. By contrast, for our own implementation, there is a clear trend towards the “quite well” option with more than half of the answers being “quite well” or “perfectly”. This is the kind of result we expected to see because making the places match the user’s preferences better was the main focus of our work (see section 3.3.2 and 3.4.3). Pearson’s correlation coefficient definitely helped to cater more heavily to the user’s preferences. And even though incorporating that coefficient was already a big step, when looking at the graph, it is also apparent that this feature can still be improved. The values for “rather not” and “quite well” could probably still be lowered some more with further refinements to the algorithm.

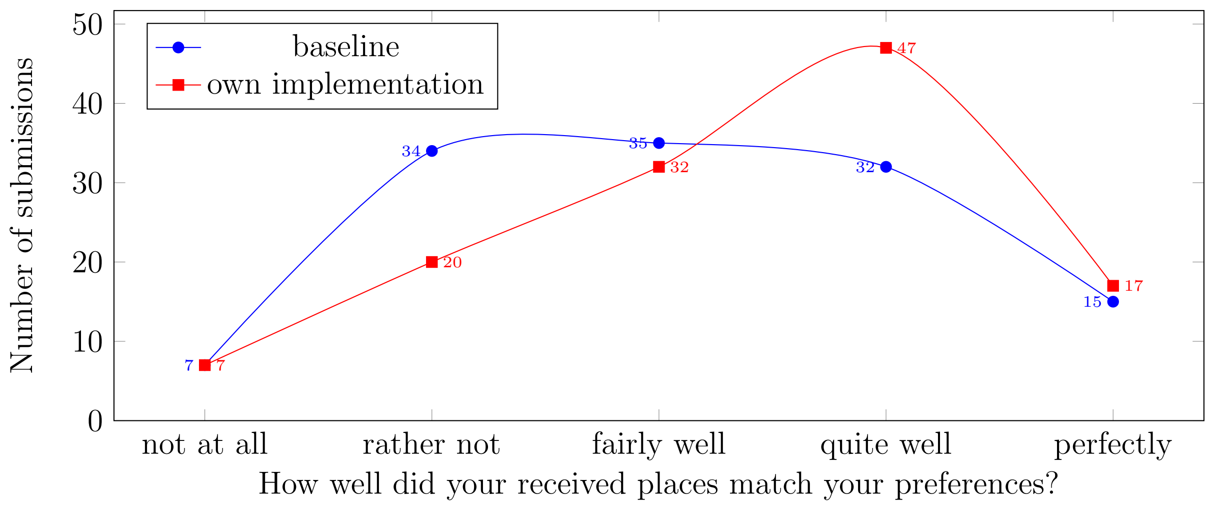


Figure 4.4: Results of the third question

Only slight improvements can be detected for the next question. We asked the users whether they would actually take this route, and gave the three options “no”, “maybe”, and “yes”. And even though the value for “no” is very clearly the lowest of the three options, the rest of the responses are pretty much equally split between the other two options. That hints at the fact that a lot of people are still indecisive about the route proposal. Concerning this question, our implementation only made minor improvements to the user rating in that the amount of people responding with “yes” increased by $\frac{53-48}{48} \approx 10\%$. The resulting graph is displayed in figure 4.5.

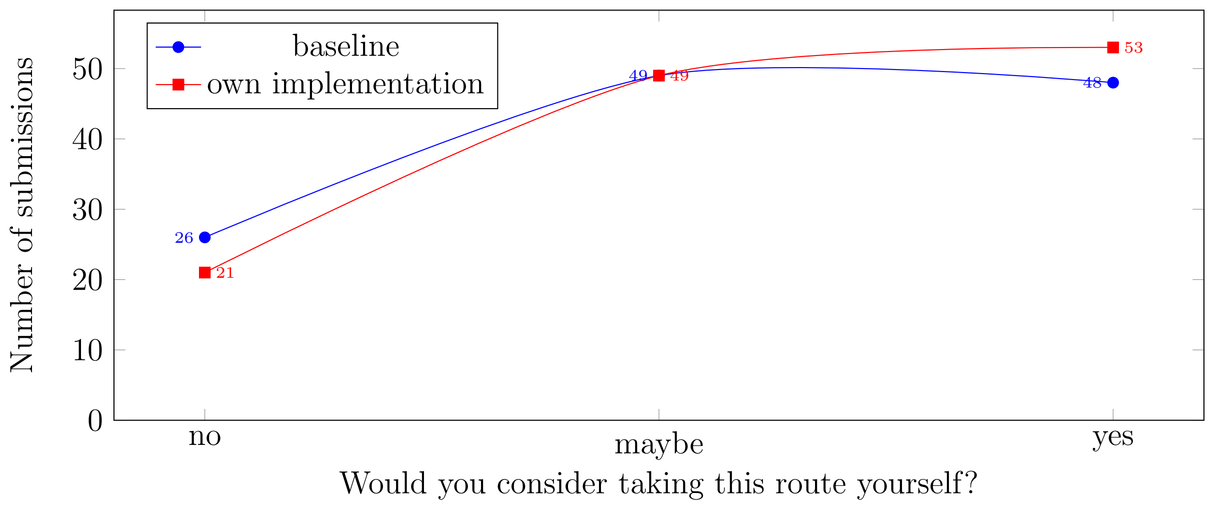


Figure 4.5: Results of the fourth question

Just like the previous question, the next one wasn’t asking about a specific detail of the paths, but about the routes in general. The question was “How satisfied are you with the overall result?”. Here we can see that our own implementation is more popular than the baseline approach, for the total amount of people that are “quite” or “very” satisfied with our implementation is $\frac{(46+16)-(38+11)}{38+11} \approx 26\%$ higher compared to the baseline (see figure 4.6). Also, the amount of submissions for the other three (worse-rated) answering options is consistently noticeably lower than for the baseline.

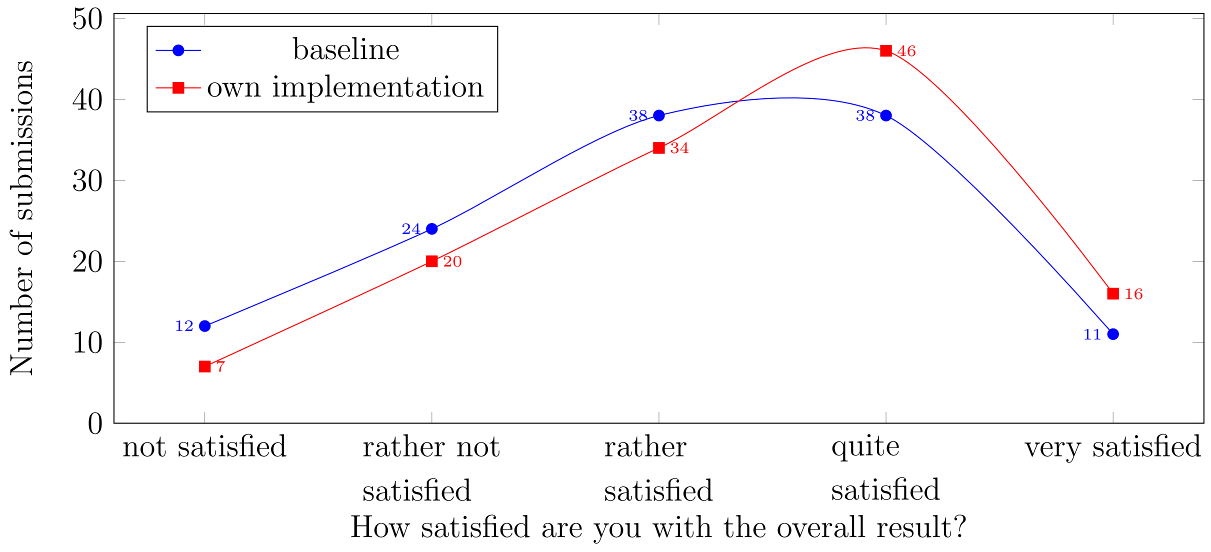


Figure 4.6: Results of the fifth question

All in all, our implementation is definitely an improvement over the baseline approach, especially with respect to how well user preferences are integrated in the path. Another way of visualizing this result can be seen in table 4.2 which shows the average user rating for each of the five questions. For that, we had to assign numbers to the response options. The first answer gets the number 1, the second gets number 2, and so on. Since question four only has three options, the numbers for the answers range from only 1 to 3; all other question answers range from 1 to 5. Now, because the optimum of the first two questions is the middle value, the perfect score here is a 3. For all other questions a higher number means a better score.

For four out of the five questions our own implementation achieves a better result than the baseline approach. As noted before, the most improvements could be made for questions three (“How well did your received places match your preferences?”) and five (“How satisfied are you with the overall result?”).

To conclude this chapter, we take a look at the standard deviation of the average user rating (table 4.3). We already mentioned above that the bell-shaped curve of the second

Question ID	**Average rating for baseline**	**Average rating for our own implementation**	**value range**	**Best possible value**
1	3.17	**3.08**	1 – 5	3
2	**3.11**	3.12	1 – 5	3
3	3.11	**3.38**	1 – 5	5
4	2.18	**2.26**	1 – 3	3
5	3.10	**3.36**	1 – 5	5

Table 4.2: Average user rating for the five questions

question is noticeably steep compared to the other curves, meaning that the standard deviation must be relatively low. In fact, that standard deviation is the lowest of all of the five questions. Only the standard deviation for question number four comes close to that value, but that is just because that question has three answering options, instead of five. All the other standard deviations are, with a value of around one, pretty normal.

Question ID	**Standard deviation for baseline**	**Standard deviation for our own implementation**
1	1.11	1.14
2	0.67	0.63
3	1.12	1.09
4	0.76	0.73
5	1.12	1.08

Table 4.3: Standard deviation of the user rating for the five questions

4.2.3 Study of user preferences

Besides analyzing direct user feedback, observing user behavior might also help making improvements to the application in the future. An interesting aspect of that is investigating the choice of user preferences. Table 4.4 shows the average user rating and the accompanying standard deviation for all six categories. Without doubt, the most popular category was “Outdoors & Recreation”. Experience has shown, however, that in this particular category there are often only few venues. Seeing that it is so popular among users, it might be worth investigating how to increase the amount of places in that category.

In contrast, the “Shopping” category was least popular. This could be due to the fact, that on a city trip, people want to do things that they can’t or don’t normally do in their home town, like visiting sights, doing recreational activities, or eating at a local restaurant. Apparently, for the 123 testers of our website, this does not include going on long shopping tours or visiting bars and nightclubs.

Category	**Average user rating**	**Standard deviation**
Sights & Museums	3.20	1.50
Night Life	2.33	1.67
Food	3.07	1.48
Outdoors & Recreation	3.48	1.30
Music & Events	2.92	1.51
Shopping	1.88	1.48

Table 4.4: Average user rating for each category

Figure 4.7 shows a detailed graph of the preference distribution for all categories. Note that the order of the categories for preference 0 is exactly the reverse of the order of the categories for preference 5.

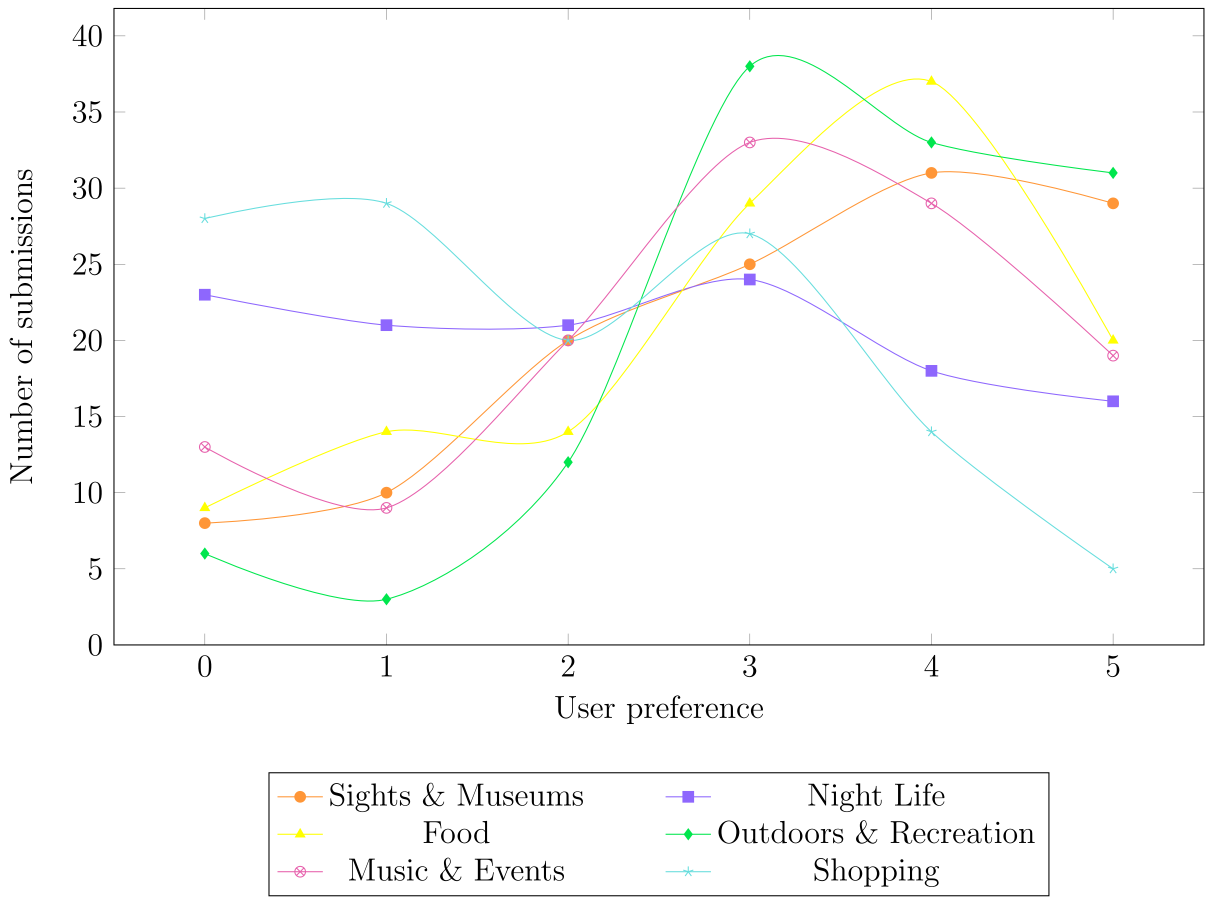


Figure 4.7: User preferences for all categories

4.2.4 Textual responses

Users had the option to anonymously submit text comments along with their rating of the routes. 33 participants used this feature and shared their thoughts with us. A lot of comments were very positive and supportive. Some users said for example that they liked a particular route really well, others liked the idea of the application itself or the website design.

There were however quite a few critical comments as well. Many of them brought up good points to further improve the application, like the following comment.

> "I have the feeling, that the system should not recommend more than 2 restaurants without explicitly marking them as alternatives, just because I can't eat in all 10 suggested one's. I think it would be good to make this dependent on the category since for shopping 10 shops might be a good fit. Maybe some additional UI components could help you to get over the cold start problem: Check which restaurants are on the way and rank them by Fsq Checkins, then ask the user 2-3 crisp questions (Eg. Thai or Italian, Cheap or Pricy) and throw out what he did not take."
>
> *Anonymous*

The fact that there are too many restaurants on the route is actually the most commonly named point of criticism and is definitely a good point. Beyond that, participants criticized the amount of venues on specific paths (in both directions), bad time or cost estimations, or that their route didn't contain the places they would have expected. Several people also said that the routes were very similar to each other. These are all valid points and can all be subject to further improvements of the project.

One commentator left a particularly negative remark saying that the project is "a useless waste of time" and "enslaved by the spirit of the age". However, people abusing the anonymity of the Internet to leave hateful and insulting comments must be anticipated when providing anonymous ways of engaging on a website and shouldn't be taken too seriously.

4.3 Conclusion

All in all, the user evaluation was a great success. We did get the results we were expecting which is that our own implementation manages to correlate the user preferences better with the places in the route. And compared to the baseline approach, users are noticeably more satisfied with our improvements. This fact is furthermore supported by the last feedback question that directly asked the participants which route they liked better. The most people answered that they liked our own implementation the most (see figure 4.8). Again, while rating the routes the users of course did not know which color corresponds to which implementation.

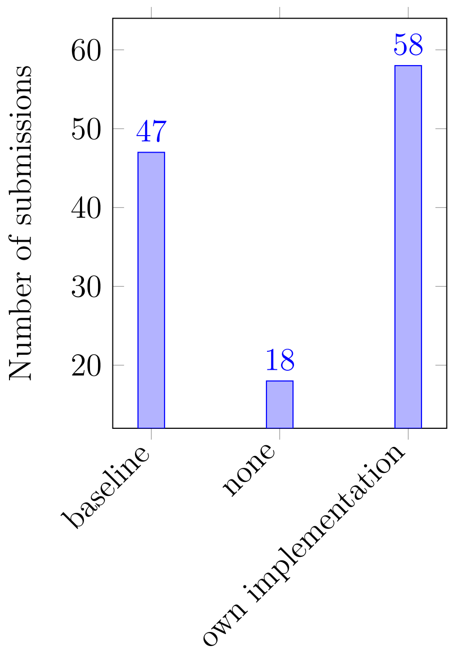


Figure 4.8: Results of the question asking for the user's preferred route

Besides the user's ratings we also received many valuable suggestions for further improving the application. In the next chapter we present ideas to continue this project in the future.

Chapter 5

Summary and future work

In this chapter we will give a short summary of what we did and will give ideas for future projects to extend our work.

5.1 Summary

Here we present a summary of both the project we developed and of the thesis we wrote.

5.1.1 Project summary

We created a website that lets the user plan a short city trip. The user rates six predefined categories, chooses whether they want to set a time and budget limit for the trip or not, and enters two places within one city on a map. Using this information, the system computes two paths between the two points consisting of as interesting venues as possible. These venues include sights, restaurants, shops, and more, depending on the preferences of the user. One of the two displayed paths is a preexisting baseline approach we adopted and the other route represents our own implementation that is influenced by the baseline but implements several improvements. After looking at the paths on the map the user can rate both routes based on a couple of questions on the website and submit their feedback. To help them decide on their answers, two overview tables listing all the places of both routes are displayed on the screen. All in all, the application consists of approximately 3,300 lines of code (not counting style sheets and markup code).

5.1.2 Thesis summary

In the thesis, we first presented all the frameworks and technologies we used for the project, most importantly the Google Maps API and the Foursquare API. We also introduced the front end frameworks Bootstrap and jQuery, and the back end technologies PHP and PostgreSQL.

We gave an overview of all the source files and their function in the application, before we got to the description of the algorithms. That chapter was split into four parts. We began by presenting the baseline place exploration algorithm, showed a way to increase the amount of places retrieved from Foursquare, and improved the classification of places. After that, we described the constraint-free algorithm and implemented Pearson's correlation coefficient to correlate better between the user's preferences and the places shown in the map. We did the same thing of the constraint-based algorithm, and also made better estimations of time and cost, compared to the baseline approach. Finally, we presented the algorithm used to display the results on the website which circumvents the limitations of the Google Maps API.

At the end, we described how the user study was conducted and analyzed the feedback of the 123 participants. Most notably, evaluation of user ratings revealed that our implementation significantly improved how well the places in the path match the user's preferences due to the incorporation of Pearson's correlation coefficient in the algorithm. Also, users were overall more satisfied with our own algorithm compared to the baseline approach. Additionally, valuable information about the user satisfaction was obtained by analyzing user ratings and direct textual feedback. These suggestions are basis for the final section of this thesis, future work.

5.2 Future work

Many users complained about the fact that start and end point can at most be five kilometers apart from each other. They want to use the application to plan trips between cities as well. Further investigation is necessary in order to be able to lift that restriction and at the same time maintain the quality of the results. Another way to weight the entertainment value and the distance of the venues has to be thought of, since for trips between cities the distance between places drastically increases while the maximum rating stays the same. Also, other means of transportation have to be incorporated when pursuing this idea.

Other improvements can be made concerning the places on the route. Often, users said that they rated the “Food” category highly but didn’t expect that many restaurants on the route. Additional user input might help decide which restaurants to recommend to the user, as there are so many different types. Instead of showing them all sequentially on the route, they could also be presented as alternatives for the user to pick the ones they like. Along the same lines goes the suggestion to try not to show more than one or two places of the same category directly after each other which also improves the diversity of the route.

Incorporating external circumstances could be another future research area. That includes a couple of things, like weather conditions or the time of day. If the user requests a route in the evening, more night life venues should be considered, whereas during the day, the user might want to see more sights. Similarly, if it is raining, more indoor places should be suggested. Of course, there should also be an option to turn these implicitly collected external factors off, in case the user doesn’t want to walk the route right now, but plans it for some other day in the future.

Another way to continue this project is to experiment with more user input options. For example, by letting the user set their walking speed to slow, normal, or fast, the constraint-based algorithm can compute a more accurate path. Also, letting the user adjust the suggested path to their own needs could improve the results. After the initial route has been computed, the system could for example let the user change individual time or budget constraints, or even let them delete places entirely from the path. Based on this input, the recommender system then computes a new route that fits the user’s preferences better. During the recalculation the system might even be able to add new places to the path based the user’s feedback for the first route. After very few iterations, a near-optimal route can be found this way.

Lastly, machine learning techniques could be used to make the system improve itself over time by analyzing the feedback submitted on the page. The basic idea is that if a user has rated a certain aspect of a route badly (e.g. if it is too long or there are too few places, etc.), the system should accordingly adjust that aspect of the route for similar requests in the future. To evaluate the algorithm, the user ratings could be compared over time to see if there has been an improvement in user satisfaction. Implementing such a learning system, however, is definitely not a trivial task.

List of Figures

List of Tables

Bibliography

[Apac] Apache Software Foundation. "Apache HTTP Server Project". https://httpd.apache.org/.

[Deve] Deveria, Alexis. "Can I use ... Geolocation". http://caniuse.com/#feat=geolocation.

[Foura] Foursquare Labs, Inc. "Authentication". https://developer.foursquare.com/overview/auth.

[Fourb] Foursquare Labs, Inc. "Explore Recommended and Popular Venues". https://developer.foursquare.com/docs/venues/explore.

[Fourc] Foursquare Labs, Inc. "Foursquare Category Hierarchy". https://developer.foursquare.com/categorytree.

[Fourd] Foursquare Labs, Inc. "Place Ratings". https://support.foursquare.com/hc/en-us/articles/201065150-Place-ratings-.

[Foure] Foursquare Labs, Inc. "Rate Limits". https://developer.foursquare.com/overview/ratelimits.

[Googa] Google Inc. "The Google Directions API". https://developers.google.com/maps/documentation/directions/intro#Limits.

[Googb] Google, Inc. "Usage Limits and Billing". https://developers.google.com/maps/documentation/javascript/usage.

[Hur] Hur, Min. "Bootstrap Toggle". http://www.bootstraptoggle.com/.

[Ilti 14] H. Iltifat. *Generation of paths through discovered places based on a recommender system.* Master's thesis, TU München, Germany, Oct. 2014.

[jQuea] jQuery Foundation, The. "jQuery". https://jquery.com/.

[jQueb] jQuery Foundation, The. "jQuery browser support". https://jquery.com/browser-support/.

[Kemp] Kemp, Kyle J. “Slider for Bootstrap”. http://seiyria.com/bootstrap-slider/.

[Meli 12] J. Melià-Seguí, R. Zhang, E. Bart, B. Price, and O. Brdiczka. “Activity Duration Analysis for Context-aware Services Using Foursquare Check-ins”. *Proceedings of the 2012 international workshop on Self-aware internet of things*, pp. 13–18, 2012.

[Micr] Microsoft Corporation. “Bing Maps API”. https://www.microsoft.com/maps/choose-your-bing-maps-API.aspx.

[Open] OpenStreetMap Foundation. “OpenStreetMap API”. http://wiki.openstreetmap.org/wiki/API.

[Open 13] Open Web Application Security Project Foundation. “OWASP Top 10”. https://www.owasp.org/index.php/Top_10_2013-Top_10, 2013.

[PHP a] PHP Group, The. “htmlspecialchars”. http://php.net/manual/en/function.htmlspecialchars.php.

[PHP b] PHP Group, The. “pg_query_params”. http://php.net/manual/en/function.pg-query-params.php.

[PHP c] PHP Group, The. “PHP: Hypertext Preprocessor”. http://php.net/.

[Post] PostgreSQL Global Development Group, The. “PostgreSQL”. http://www.postgresql.org/.

[Twita] Twitter Inc. “Bootstrap”. http://getbootstrap.com/.

[Twitb] Twitter Inc. “Bootstrap browser and device support”. http://getbootstrap.com/getting-started/#support.

[W3C] W3C. “Geolocation API Specification”. http://dev.w3.org/geo/api/spec-source.html.

[Weisa] Weisstein, Eric W. “Correlation Coefficient”. http://mathworld.wolfram.com/CorrelationCoefficient.html.

[Weisb] Weisstein, Eric W. “Sample Variance”. http://mathworld.wolfram.com/SampleVariance.html.