Overlapping Partial Trips for Car-Sharing applications

Sascha Feldmann

Matrikel-Nr. 547307

INDEPENDENT COURSEWORKS

eingereicht am
Masterstudiengang

Internationale Medieninformatik (Master)

an der Hochschule für Technik und Wirtschaft, Berlin

im August 2015

Declaration

I hereby declare and confirm that this coursework is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hochschule für Technik und Wirtschaft, Berlin, August 28, 2015

Sascha Feldmann

Contents

Chapter 1

Introduction

The context of this work is an existing Car-Sharing application which was developed by Martin Schultz and the author during their bachelor program at Beuth Hochschule für Technik, Berlin. The existing application was originally developed in 2013 as multi-tier web application using PHP Zend Framework 2. In 2014/15, the author decided to develop a completely new J2EE-based web application together with Marco Seidler.

The application allows car owners to offer trips for potential passengers. Therefore, the driver has to configure the trip she/he's planning for some date and time. She/He needs to configure the exact route she/he plans to follow using a dynamic map which is the central interaction element. The driver is asked to define an overall price for the whole trip which mostly includes prices for gas and wastage of her/his configured vehicle. Passengers can find matching trips by a map-based search engine that evaluates the coordinates of the passenger's start and end location against all the trip paths that take place in her/his prefered date range. The system will present a calculated price based on the overall price that the driver entered. It takes care of the distance of the passenger's partial trip. The underlying algorithm will be explained more detailed in chapter 2.

So, the current application covers the following use case: a passenger wants to get from Hannover to Cologne. She / he enters her / his prefered travel time, the maximum distance that she / he is able to travel by her or himself to get to the meeting points and the system matches a passing trip from Berlin to Bonn. This trip is close enough to the passenger's start and end locations. The final meeting locations have to be clarified by informal communication between both driver and passenger. Thus, the current application already offers algorithms to find partial trips on *single* routes offered by drivers. Compared to other Car-Sharing platforms, this capability appears to be an overvalue because most platforms require driver and passenger to have exactly the same start and end location or at least an explicitly defined waypoint.

The current application therefore allows to load cars more efficiently in the sense of ecologies and economics. The goal of this work is to extend the existing algorithms to find partial trips that overlap multiple routes of different drivers. Following the example above, the passenger would be able to find an intersection: he could take a trip from Hannover to Düsseldorf, for example, where another driver offers a trip to Frankfurt by passing Cologne.

An overlapping partial trip algorithm could increase the efficiency of loading cars. On the other hand, it could increase the mobility of users who need to get from certain start to another end location. Thinking of the close-meshed road network of Europe this would also simplify the passenger's search because she/he doesn't need to try out a couple of start and end points in order to find a matching trip.

In chapter 3, the technology requirements and potential algorithms that are mostly based on graph concepts are discussed. At the end, those are compared by their complexitiy and applicability on the stated problem.

The results are an input for the final concept that is presented in chapter 4. Based on this, a working solution was implemented. The implementation is described in 5. It is demonstrated by screenshots of the extended web application in chapter 6.

The found solution needs to be discussed in matters of performance in real-time web applications. Chapter 6 therefore collects several measurements of the developed algorithm runtimes and complexity.

In the final chapter 7 the solution will be reviewed. It is therefore discussed whether the algorithm produced the expected results in an optimal runtime. Also, possible alternative solutions that were briefly introduced during the research are discussed.

In summary it can be said that the algorithm of finding overlapping partial trips are an overvalue within the problem domain of Car-Sharing applications. But one could also think of applying the solution on any other traffic network problem - such as public transport or forwarding companies.

The source code of the application is open and can be found on Github [1]. Since the submission of this coursework, the source codes were written by Sascha Feldmann and Marco Seidler only. All extensions within the scope of this coursework were done by Sascha Feldmann only.

[1] https://github.com/sasfeld/JumpUpReloaded

Chapter 2

Current State

In this chapter the current state of the car sharing application will be explained. First, the application will be introduced from the user's perspective. Then, the application's architecture will be sketched shortly. Afterwards, the current founding algorithm to find partial trips by one driver will be presented. Finally, the current solution will be concluded to the extension to find overlapping partial trips of multiple drivers.

2.1 User Experience

A user can either act as driver or passenger. In this chapter, the tasks that are necessary for the matching algorithm will be explained separately.

2.1.1 Offering a trip

If a driver wants to offer a trip, she/he needs to enter a start location, final destination, departure and estimated arrival date and time, an overall price for his trip and the number of seats he want's to offer.

The locations inputs are auto-completed by making use of Google Map's Directions service [1]. If both were entered, Google's route service is used to get the trip's path as a list of coordinates. This list of coordinates can be large, e.g. 226 on the default response for "Berlin -> Bonn" (596 km). Looking at the ratio of coordinates and trip distance (226/596), you can approximate that the DirectionsService returns one coordinate per each two to three kilometers.

Therefore, the client-side javascript was extended to reduce the number of path coordinates. The approach is shown in figure 2.1 . First, an empty list is created. Then, it will be iterated through all returned path points. In

[1] The decision to use GoogleMap was met because of the large capabilities of the Directions Service. Currently, Google allows 25,000 free map loads per day ([**GoogleMapLimits**]).

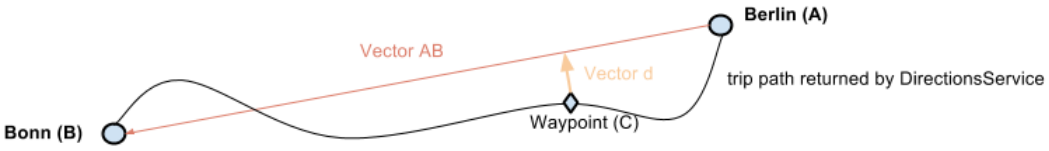

Figure 2.1: Explanation of strategy to reduce stored waypoints

each iteration, we build two vectors: vector AB between the previous point and final destination and the vertical vector between the current path point and AB to determine the distance. If the distance exceeds a specified limit, it is added to the previously empty list, so the list will grow to only contain those path points that are far enough from the the previous points. Fig. 2.1 shows the state in the first iteration.

After the map processing, the data is posted to the application backend and stored in a new trip entity.

2.1.2 Searching for trips

When the passenger looks for trips, she/he first needs to apply the following filters in a form:

- Filling in her/his start location and final destination.
- Prefered date and time range.
- Price Range in euro.
- Distance in kilometers that she/he's able to travel by her-/himself in order to meet the driver on her/his route.

The input of the locations is supported by auto-completion so that the exact location is matched. The user can fill in a location name like "Berlin". Google Map's location service will deliver possible exact matches, like "Berlin, Germany" which refers to the capital of Germany. Internally, the coordinates of the matched locations will be attached to the filter form.

Now, the filter values form can be sent to the application's backend. The backend processes the query for matching (partial) trips as following:

- Trigger an HQL query [2] to receive trips that take place in the given date range.
- Delegate to the instance of the NearbyTripsFilter class which will filter the list of trip entities for those that pass the passenger's start location and destination within the prefered distance.

[2] Hibernate Query Language - special object-oriented syntax of the ORM provider which will be transformed into a SQL query

2.2 Algorithm

The implementation of NearbyTripsFilter realizes a Greedy algorithm.

Therefore, it will be iterated over the given trips list. On each iteration, the trip's path coordinates will be checked against those of the passenger's start location and destination. For each waypoint, the distance to the passenger's locations will be calculated using the "haversine" formula to get the shortest distance by taking care of the earth's sphere characteristics which is explained in chapter 3. If the formula returns a value that is less than or equal to the passenger's configured distance, the waypoints loop will be exited and the trip is added to the list of matching trips.

2.3 Complexity

The problem size of the described Greedy algorithm is given by two inputs: the number of trips (N) returned in step 1 and the sum of the numbers of waypoints per trip (K).

$$\Theta(N) = N * K$$

The notations show that the complexity grows if either the number of trips from step 1 or the sum of waypoints increase. In theory, the waypoints sum has the highest risk to grow. This is due to the nature of long-distance trips. For a trip from Berlin to Bonn, 134 waypoints will be saved for example.

2.4 Architecture

The car sharing application follows the multi-tier approach of JEE - platforms. As shown in figure 2.2, the logic to recieve the trip and passenger's start and destination coordinates is nested in the Web Browser on the Client tier. Here, a javascript querying Google Maps Direction Service is triggered and the results are given to the backend application on the Web Tier. The backend application delegates to the storage or query services nested on the Business Layer. So the business layer contains the application's most important algorithm: finding matching trips for a passenger's query.

For future extensions, this core logic can easily be reused. You could think of a mobile app that communicates to a REST API on the Web Tier layer which would also delegate to the same business layer as the existing web application for example.

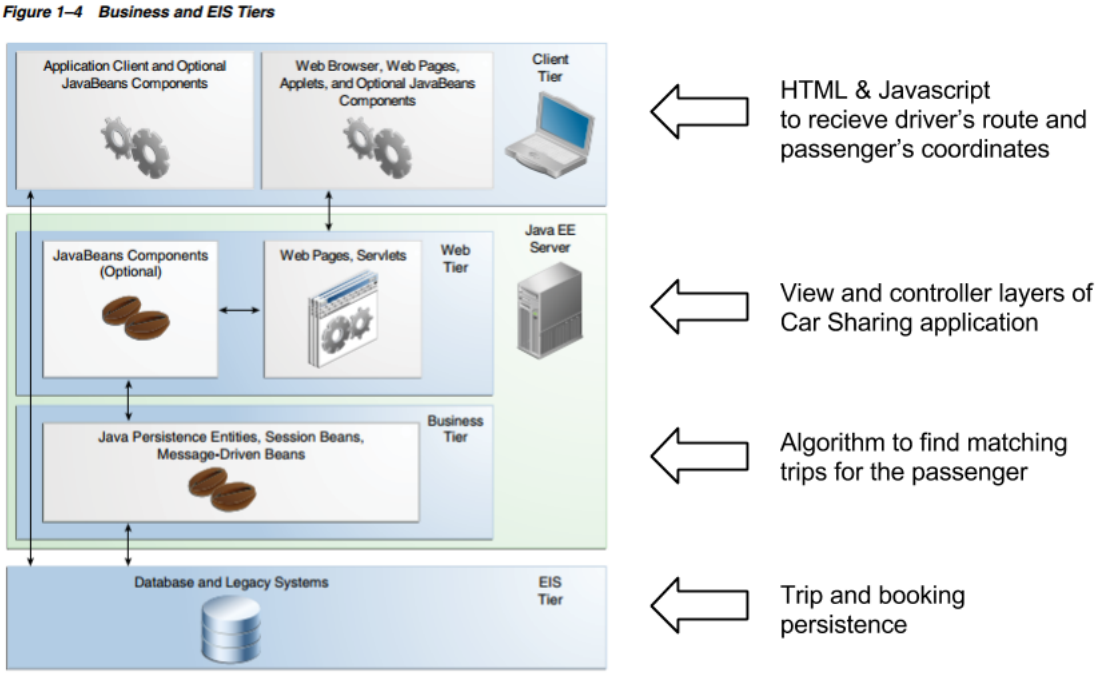

Figure 2.2: car sharing application architecture, based on J2EE 7 tutorial multi-tiers explanation [**JEE7Tutorial**], p. 1-8

2.5 Goals of this work

The existing capabilities - finding single offered trips - should be extended by also finding crossing partial trips. Therefore, the introduced algorithms will be used as foundation. As seen in the previous chapter, the extensions will mostly take place in the business tier where the main algorithm is nested.

With this work, the solution should be found by comparing different possible algorithms and check whether they are applicable in the car sharing application.

Chapter 3

Fundamentals

To understand the concept of extending the existing application it is necessary to have basic knowledge about the underlying JEE platform and basic graph theory. In this chapter, current approaches to find optimal routes that are based on partial overlapping are sketched.

3.1 Haversine formula

In the application the algorithms make use of the haversine formula to calculate the distance of two points considering the earth's sphere characteristics. If the euclidian distance would have been used by thinking of the earth of a not curved surface, it would have been necessary to use a projection algorithm to get the curved coordinates and therefore the real distance. So the haversine formula allows a performant solution for this problem. In computer science, it was used to extend the standard formula given in spherical astronomy books to get the angular separation of one star from another to be more satisfactory for very close objects (see. [**Haversine**]). It is defined as:

$$\text{hav s} = \text{hav } (\Delta\delta) + cos\delta_1 \cos \delta_2 \text{ hav } (\Delta a)$$

The haversine formula was first introduced around 1800. The underlying haversine function itself appeared earlier (see [**Smith**], p. 618). The name (half versus sinus) is coming from the Latin "sinus versus". The function itself is defined as:

$$\text{hav} = \sin^2 \frac{\delta}{2} = \frac{1-cos(\delta)}{2}$$

This formula is applied in an object-oriented style in our application can

be viewed in our public source code [1].

3.2 Java enterprise platform

As introduced in chapter 2 the application is based on JEE platform 7. The decision was met to have an optimized extensible and scalable architecture. The Java EE container shown in fig. 2.2 allows to store the main route finding algorithm in a single reusable component. The component should be used via well-defined interfaces. Therefore, this kind of architecture allows to distribute the algorithm over different systems when scaling gets important due to a growth of users. The main route finding algorithm class was defined as so called Enterprise Java Bean (EJB) that can be used by other components such as the Java Server Faces (JSF) web application via the Java Naming and Directory Interface (JNDI) lookup service (see [**JEE7Tutorial**], p. 1-9). In each layer of the application, single containers manage the execution:

- Web Container: manages the web page itself - we are using JSF to deliver the HTML, Javascript and CSS to the web browser client.
- EJB Container: manages our EJBs and therefore the business logic.

There isn't any client application container that is managed by the JEE platform. The car-sharing application is browser-based, so that the communication between both is based on HTTP.

JEE includes the Java Persistence API (JPA) that is primarily used to decouple the relational database and the object-oriented software. Implementations of JPA help to prevent this kind of paradigm mismatch (see [**Hibernate**], p. 1). Therefore it defines object relational mapping (ORM) metadata (see [**JEE7Tutorial**], p. 1-17).

Beside the JavaMail API and Java API for JSON processing the presented technologies are the most important for the Car-Sharing application. In the following chapters the concrete JPA implementation and JSF will be explained.

3.3 Hibernate

The application makes use of Hibernate which is an implementation of the JPA specification. The Car Sharing application doesn't need any logic in the database layer via stored procedures for example. As stated above, it is

[1] https://github.com/sasfeld/JumpUpReloaded/blob/master/src/main/java/de/htw/fb4/imi/jumpup/util/math/CoordinateUtil.java

contained in the EJB layer of the application. Hibernate was designed for applications like this ([**Hibernate**], p. 1).

The EJB layer also contains the entity classes that are annotated by JPA provided tags. Hibernate processes those annotations and transforms instances to the underlying SQL database schema itself and reverse. To query entities as during the trips request the application contains Hibernate Query Language (HQL) queries. The example of finding matching trips shown in 3.1 for a passengers contains criteria like the price and date ranges.

Listing 3.1: Criteria HQL Query to find trips

```
1 @NamedQuery(name = Trip.NAME_CRITERIA_QUERY, query = "SELECT t FROM Trip
      t WHERE"
2 + "t.cancelationDateTime IS NULL"
3 + " AND (t.driver != :passenger)"
4 + " AND (:dateFrom IS NULL OR t.startDateTime >= :dateFrom)"
5 + " AND (:dateTo IS NULL OR t.endDateTime <= :dateTo)"
6 + " AND (:priceFrom IS NULL OR t.price >= :priceFrom)"
7 + " AND (:priceTo IS NULL OR t.price <= :priceTo)")
```

Further extensions of the application like finding overlapping partial trips should take place in the EJB layer to keep the flexibility that is offered by the loose coupling of the EJB and storage layers.

3.4 JSF

The web layer consists of a Java Server Faces (JSF) implementation which is a framework for the creation of graphical user interfaces for web applications.

3.4.1 Facelets

It consists of so-called Facelets which are view scripts written in an XHTML syntax. Therefore, JSF provides different tag libraries to add view components via XHTML annotations (see [**JEE7Tutorial**], p. 8-2). The car sharing application contains custom components such as an e-Mail address field that comes in with additional validation declarations. JSF will process the special XHTML and produce pure HTML 5 before sending the response to the browser. The technology is part of the Java Web Application technologies as shown in figure 3.1 and can be used in parallel to Java Server Pages and Java Servlets. Itself is based on servlets technology.

JSF enforces to make use of the Model View Control (MVC) Pattern by clearly separating facelets and the Java Bean layer which are addressed using the Expression Language (EL). In the car sharing application, facelets are referencing Java Beans on the Web Tier which are mostly controller classes. The models are nested in the EJB business tier.

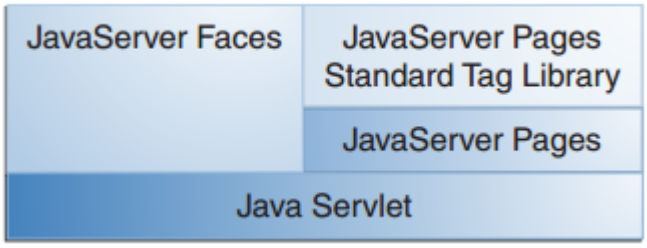

Figure 3.1: JSF nesting in Java Web Application Technologies [**JEE7Tutorial**], p. 7-3

3.4.2 Request-Response Lifecycle

Java Server Faces provides a lifecycle specification with the Execute and Render main phases ([**JEE7Tutorial**], p. 7-13). In the execution phase, each view input element will be validated for example. On validation failures the processing of the input field will be terminated which is good in terms of security. JSF simplifies the HTTP Request parameter handling, validation and processing workflow and manages the component states (e.g. in a session). The car sharing application contains validation classes in the Web Tier for each custom view element, for example. Those simply implement a JSF interface.

Figure 3.2 shows the desribed request-response lifecycle. If any validastion fails, the model values in the Java Bean layer wouldn't be updated at all. Processing would stop before the application respectivly its controller classes in the web tier are invoked. Then, an error message would be shown. This strict process protects the car sharing application from invalid inputs like injection attacks.

Further extension of the car sharing application therefore need to fit the JSF lifecycle and MVC concept so that the security, maintainability and flexibility of the web tier can be kept.

3.5 Basic graph theory

Because the problem of this work can be typically found within the graph theory, basic concepts need to be sketched shortly.

3.5.1 Definition

A graph in the car sharing application consists of offered trips which are paths in the overall directed graph of waypoints (coordinates) and edges between them. Formally, a graph is described by (see [**Krumke**], p. 7):

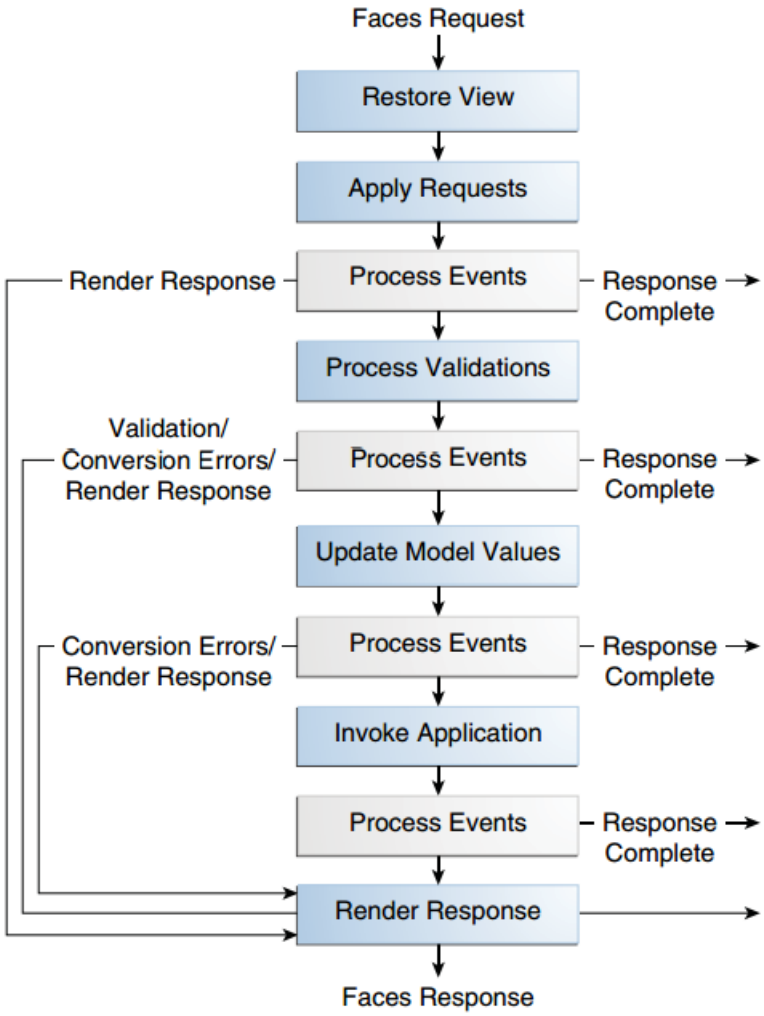

Figure 3.2: JSF Request-Response Lifecycle [**JEE7Tutorial**], p. 7-14

g = (V, R, α, ω)
V is a set of vertices
R is a set of edges
$\alpha : R \to V$ (beginning vertex of an edge)
$\omega : R \to V$ (ending vertix of an edge)
Way in G: set of (v_0, r_1, v_1,...., r_k, v_k)

3.5.2 Storage of graphs

As introduced in chapter 2, trips are currently stored unstructured as list of waypoints described by coordinates. To fulfill the task to find overlapping partial trips using graph algorithms, the data needs to be structured in order to describe a graph formally.

One possibility to store graph on the application level is to use a lightweight array structure based on the adjacency matrix. Formally, the matrix is built by (compare [**Krumke**], p. 19):

$$\mathrm{a}_i j = |\mathrm{r} \in R : \alpha(r) = v_i \text{ and } \omega(r) = v_j|$$

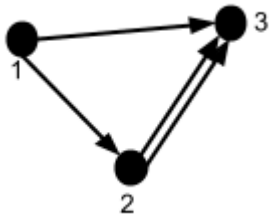

Figure 3.3: Example of a directed graph

Following this definition, the graph in figure 3.3 could be stored as the following matrix:

$$A(G) = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

Rows and columns in the matrix are representing the vertices beginning from one. The values in the cells show the number of connecting edges.

Listing 3.2: Storage of an adjacency matrix in Java

```
1 int[][] adjacencyMatrix = {
2         {0, 1, 1}, // connections from vertex 1
3         {0, 0, 2}, // connections from vertex 2
4         {0, 0, 0} // connections from vertex 3
5     };
```

In Java, this matrix should be saved as the two-dimensional array shown in 3.2. But a more common approach is to make use of an adjacency list ([**Krumke**], p. 21) which consists of sub linked lists indicating connected vertices for each single vertex. An example is shown in fig. 3.4

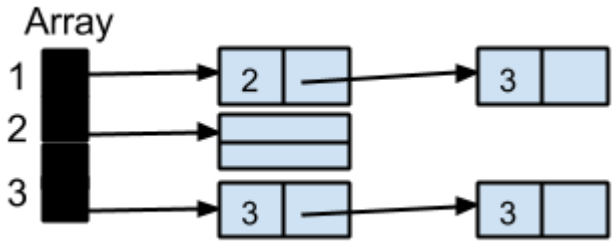

Figure 3.4: Example of the adjacency list

Therefore, the application should make use of the array adjacency list structure to process the partial overlap searching algorithm.

3.6 Shortest Path

The extension of the car sharing application opens a shortest path finding problem: now, not only matching trips that connect *directly* a passengers start and destination location need to be found, but also connecting trips in a graph structure. Therefore, we need to find the shortest path in terms of duration and distance.This kind of path problem is a Single Pair Shortest Path Problem (SPP) which can be differentiated from Single Source Path problems (SSP) and All Pairs Shortest Path problems (APSP) (see [**Krumke**], p. 167).

3.6.1 Comparison

According to [**Zhan**], the following shortest path algorithms run fastest on real road networks:

- Pallottino's Graph Growth Algorithms with two queues
- Dijkstra Algorithm with approximate bucket
- Dijkstra algorithm with double buckets

During the study of Zhan and Noon, it was stated that Pallottino's graph growth algorithm was the best for solving Single Source Path problems, while Dijkstra offers "advantages" for Single Pair Shortest Path problems (see [**Zhan**], p. 74). They explained that Dijkstra could be terminated when the destination node is reached.

The Dijkstra algorithm should be used due to the optimal runtime of $0(n^2)$. It will be explained in the following chapter.

3.6.2 Dijkstra

The Dijkstra algorithm ([**Dijkstra**]) solves the Single Source Path problem. The input is a graph G in adjacency list representation, a weight function c and a start vertex s. The algorithm works by making use of a priority queue, distances markers for each vertex and a list of visited vertices. In each iteration, the first node of the priority queue (with the minimum cost marked) will be extracted. Following the so-called "relaxation" principle, all successor vertices from the current node on will be marked with the costs of the current vertex plus the costs of the edge if the current costs of the successor are larger. The algorithm terminates if the priority queue is empty (so all nodes were visited). The distance markers on each vertex show the shortest path from the start vertex s.

The car sharing application should modify the algorithm to build a predecessor list back from the passenger's destination to the start vertex s so that a found route can be traced.

3.7 K-Shortest Path

The target of this work is to find overlaps between different paths (trips offered by drivers). It makes sense to find more than one solution. Alternative shortest paths can be found by using the k-shortest Path algorithm.

[**Yen**] proposed a method to find K shortest loopless paths in a network from an origin to a destination node. Compared to other shortest path algorithms it allows to find alternative routes which would improve the search in the car sharing application. The algorithm mostly works by the random deletion of nodes on the shortest path between two nodes (see [**Lim**], p. 1).

3.7.1 Partial overlap proposal

In [**Zhou**], a k-shortest path algorithm was proposed to the problem of a complex transit network software - a Geographic Information System (GIS). People who want to use a transit system in a metropolis often have the problem to find non-recurrent trips from an origin to a destination that contains different combinations of means of traffic. Think of getting around in Berlin: people can use the tram, bus, S-Bahn, subway, regional trains, bicycles or a car. Here, transit advanced traveler information systems (TATIS) could help people to find around more efficiently. They find connections between different means of traffic by minimizing the "cost of the whole path" ([**Zhou**], p. 1).

The stated advantage of the k-shortest path approach is that it "can provide several alternative paths" ([**Zhou**], p. 2) and therefore fit the nature of a human's need: offering alternative solutions and not only restricting to one shortest path.

In the study, a new method called "partial overlap" was developed. First, a road layer and a transit network graph were created. Then they were combined to a new topological structure by finding nearest intersection nodes on the road layer for each public transport stop in the transit layer. Partial overlapping here was meant to be the overlapping between each path and the travel costs (see [**Zhou**], p. 5).

This "partial overlap" approach is interesting for the finding overlapping partial trips in the Car-Sharing application because it takes care of the travel costs during the connection. A passenger will have to switch cars when an intersection between different trips was found. The costs here would be connecting times and distance between the intersection locations, for example.

3.7.2 Path overlap in road network

The study of [**Lim**] was focused on applying the path overlap to real road networks and therefore interesting for the extension of the car sharing application.

The authors stated the weakness of too much heterogeneity between the alternative paths coming from the k-shortest path algorithm. Therefore they proposed an algorithm to build a new path by evaluating the degree of overlapping in terms of travel costs ([**Lim**], p. 2). This could help to offer optimal alternatives for matched overlapping partial trips in the car sharing application. For example, the result list could deliver more widespread connecting times and distances for intersections.

Another interesting outcome is the support of so-called "penalized turns" ([**Lim**], p. 3). In real road-networks, those can be restricted turnings for example. Penalized turns could be applied in the car sharing application to restrict intersections in trips: the driver could not want to stop at some location where the passenger needs to change the trip, for example.

3.8 Summary

The researched algorithms should be used to extend the car sharing application by the overlapping partial trips functionality. The extension must follow the current application structure and fit into the JEE architecture.

Chapter 4

Concept

The researched algorithms are the foundation for the concept of the extension in the car sharing application.

4.1 Requirements

The application should be extended by finding overlapping partial trips. This allows the passenger to find trips that are crossing each other. If a passenger wants to get from Berlin to Cologne for example and there are separate trips from Berlin to Hannover and Hannover to Cologne, she/he will be able to book both of them. In a first step of the extension, the existing HQL query to get all trips that take place within the passenger's prefered date range should be used. In the future, this query might be important to increase the application's performance once multiple partial trips are allowed because the number of selected trips is a dominating factor for the algorithm runtimes.

4.1.1 Functional requirements

The following requirements must be fulfilled.

- The passenger is able to find multiple (partial) crossing trips that bring her/him to his destination.
- The found multiple trips should be the shortest connection in terms of distance and duration.

The following features should be available:

- The passenger should be offered alternative routes across multiple (partial) trips so she/he can decide by his / her own how to get to the destination.
- The found multiple (partial) trips should be displayed in the existing map in the browser.

4.1.2 Technical requirements

Technically, the following limitations are important:

- The extension fits into the JEE architecture by encapsulating the extensive business logic in the EJB layer, defining controllers and JSF views in the web tier.
- The solution makes use of a graph structure, in the best case by making use of an adjacency list. Therefore, a transformation of the currently unstructured data is necessary. The graph structure represents all trips that take place in the passenger's prefered date range.
- The Dijkstra algorithm is used to find the shortest path across multiple (partial) trips so that a good runtime is gained.
- K-Shortest path algorithms *can* be used to offer alternative shortest paths.
- The used shortest path algorithms should be implemented against well defined API methods so that they are exchangeable.

4.2 Technical process

The stated requirements are pictured within the process shown in figure 4.1. The current algorithm in the application's EJB layer to find direct trips respectively trips that are passing the passenger's origin and destination will be left untouched. If no direct trip was found, the new algorithm to find overlapping partial trips will get active. Then, all found trips will be transformed into a graph structure by making use of an adjacency list. The adjacency list will be the input for the shortest path algorithm which will be an Dijkstra implementation first.

4.2.1 Extension in existing trip query service

The current trip query service method to find matching trips for given criteria needs to be extended. Figure 4.2 shows the entities Trip, Booking and User. Those can be left untouched, so the Hibernate persistence doesn't need to be extended.

Currently, the service returns an instance of TripQueryResult which consists of multiple SingleTripQueryResult instances. The frontend relies on this data structure so it should still be returned if direct trips were found so that no adjustments in the frontend javascript are required for the existing use case.

If no direct trip was found, the service should return a specialized TripQueryResult which structures found overlapping partial trips in another way. This specialized instance should consist of an instance of SingleOverlappingPartialTripsQueryResult or similar which references the overlapping

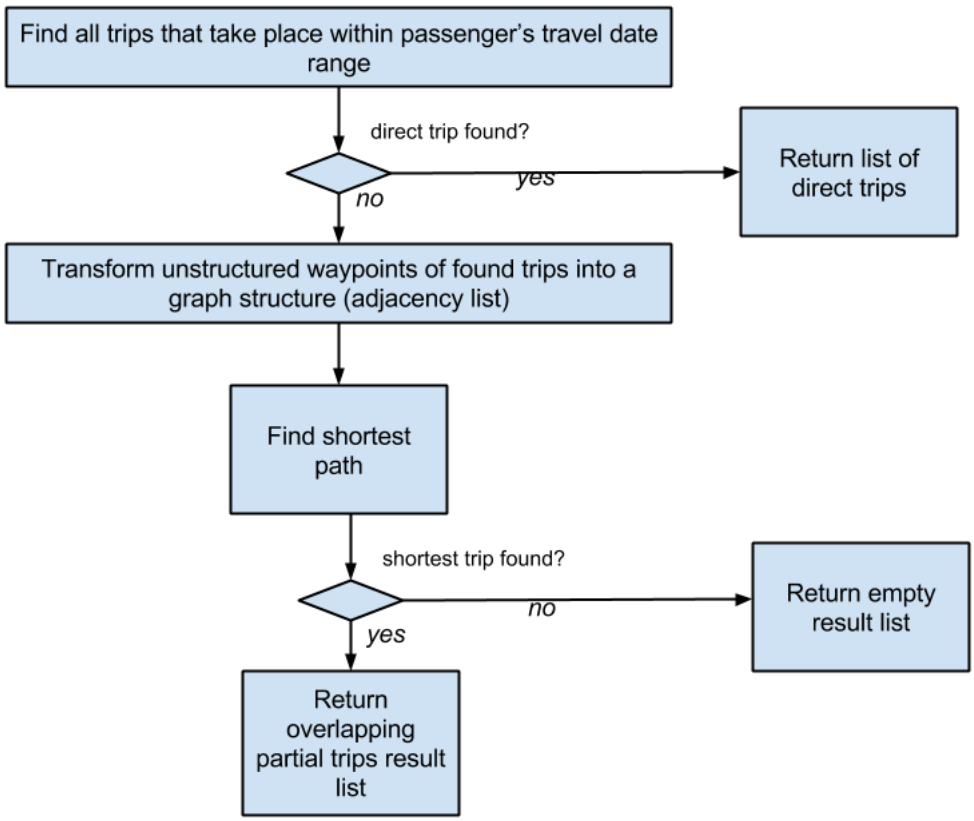

Figure 4.1: Data flow of extended find trips process in the Trip query service (EJB-Layer)

trips and contains the intersection data such as longitude, latitude and optionally connecting time.

4.2.2 Graph structure

The current waypoint data of trips is not structured. As shown in 4.2, the path of a Trip entity is stored in a property *overviewPath* of type String. This is a comma-separated list of latitude, longitude pairs. Each pair can be seen as a node in the overall graph of trips that pass the passenger's origin or destination in his or her desired travel date range.

The transformation into the graph structure reveals one problem: the overviewPath coordinates are very fine, the probability that a coordinate is hit by more than one trip is therefore low. There might be two trips which

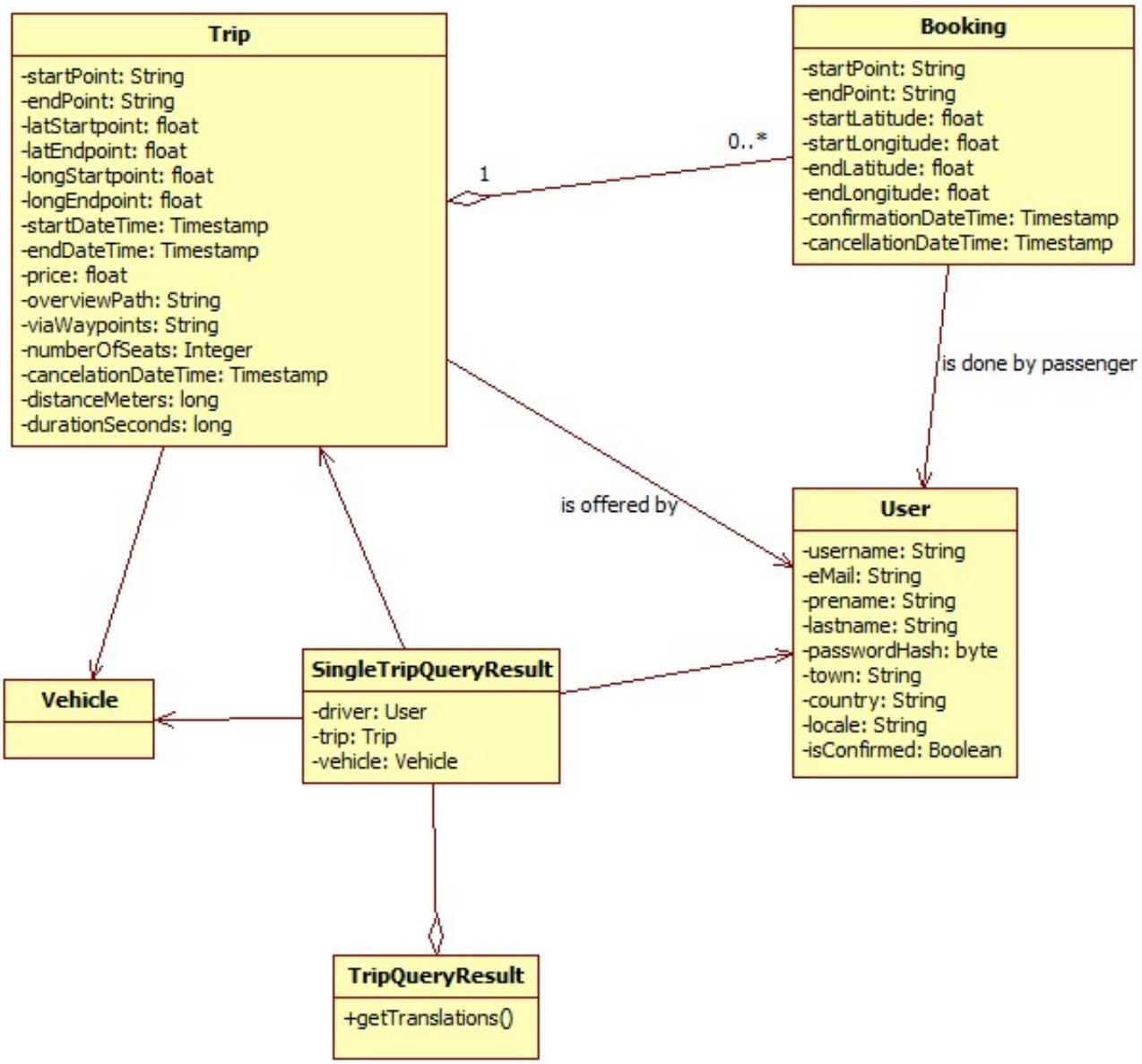

Figure 4.2: UML of basic models that are important for the extension

visually cross somewhere in Hannover, but the stored coordinates might differ even if the distance between them is less than 50 metres, for example.

To solve this problem, the coordinates should have some kind of tolerance. One solution is to round a coordinate down to a certain amount of digits after the comma before it is stored as graph node in the adjacency

list. Doing this, near coordinates would be rounded down to the same value and open up an intersection between two or more trips.

The prepared coordinates can be stored by making use of the existing Coordinates class in the adjacency list. Coordinate instances in the graph should be unique: during the creation of the graph, a new instance of coordinate should never be created if there's already an instance with the same latitude and longitude values. Otherwise, the shortest path algorithm won't work because it relies on the referential use of vertices.

4.2.3 Frontend extensions

There are no changes required if direct trips were returned from the Trip query service. If overlapping partial trips were found, the given JSON response handling needs to be extended. Therefore, the model *SingleTripQuery* result needs to be replaced by a polymorphistic structure of possible results (e.g. direct trips versus overlapping partial ones). The partial trips should be shown on the map and in the existing result list that is presented to the user. The intersection of the the overlapping trips should be visible to the passenger.

4.3 Summary

Given the must and should have criteria, an UML structure with clear points of extension and a basic concept for the graph transformation, a first solution can be implemented.

Chapter 5

Implementation

The implementation follows the concept introduced in 4. In this chapter, the extensions that were required to offer multiple partial trips are explained.

5.1 Graph implementation

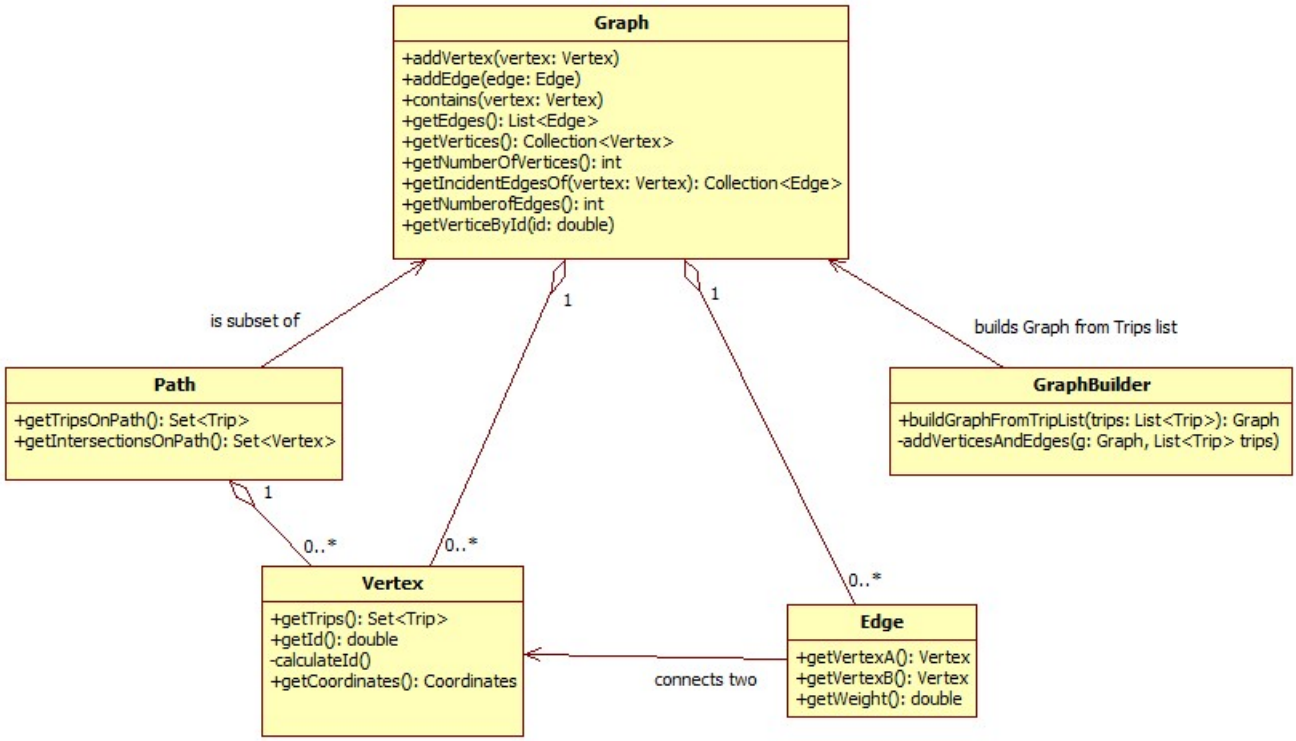

Figure 5.1: UML of the implemented Graph structure

The waypoint data was structured into a Graph model shown in figure 5.1. Therefore, the following representing Java classes were introduced:

- Graph: a Graph consists of Vertex and Edge instances. It basically capsulates the vertex and edge structure by making use of an adjacency list. It therefore allows fast access to incident edges of vertices by shortest path algorithms, for example. The graph adds a constraint for the uniqueness of vertices in the graph (e.g. same IDs). It will throw an exception if a vertex is already contained.
- Vector: a Vector is bound to a Coordinates instance which were already used before the extension. The method *getId()* will trigger the calculation of a double value that acts as unique identifier. The calculation helps to solve the connecting trips probability problem. It is explained in chapter 5.4.
- Edge: an Edge connects two Vertex instances with a specific weight. The weight is a double value so the shortest path algorithms can be more precise. Usually, the weight is the distance between the two vertices in kilometers.
- Path: As per definition, a path is a subset of vertices of a graph. Therefore, a Path implementation consists of a sequence of Vertex elements and also offers convenience methods that fit the current car sharing application, e.g. to get all the Trip instances that are contained within a path.

Those classes were fit to work with the current modelling, e.g. Vertex instances basically are bound to a Coordinates object. The transformation into this graph structure is explained in the next chapters.

5.2 Selecting trips

The existing HQL to select the trips was left untouched. It was introduced in chapter 3 by listing 3.1. Hibernate will return a list of Trip instances as result. Then, the existing algorithm to find *direct* trips will be triggered. If no results were found, the graph structure will be built and the shortest path algorithm will be triggered to find *overlapping partial trips* as explained in the following chapters.

5.3 Structuring process

An instance of the class *GraphBuilder* (figure 5.1) is responsible for structuring the waypoint data into the introduced graph implementaton. The public method simply takes a Trip instance and returns the builded graph instance. Therefore, it works as following:

- Create a vertex for the start point of the trip.

- Iterate over the trips' waypoints (instances of Coordinate) in the order of the geographical route. Create Vertex instances for each and connect them sequentially by creating Edge instances. Calculate the weight by making use of the existing helper method *calculateDistanceBetween(Coordinates coordinates1, Coordinates coordinates2)* using the *Haversine* formula.
- Create a vertex for the destination of the trip. Also create an edge between the last created vertex and the new destination by also calculating the distance using *Haversine*.
- Keep uniqueness of vertices: If a vertex needs to be added which has an ID that already exists in the graph structure, do not add a new one but use the existing one. This means that the vertex acts as intersection point between overlapping trips.
- Store referring trips on each vertex: it is very important to keep a reference between trip and vertex, so for each vertex a reference to the trip will be added into the set of references. This allows to identify intersection waypoints (the overlaps) between the trips.

Now, a graph structure was created that will be used by shortest path algorithms.

5.4 Intersection probability

The calculation of the Vertex ID solves the problem of the very fine coordinates that are stored on trip instances as described in chapter 4.2.2 by rounding the coordinates to 4 digits after the comma. The rounded off values are then sumed up to an ID. This common procedure for different waypoints allows to control the probability of finding connecting trips.

The calculation is shown in listing 5.1.

Listing 5.1: Calculation of Vertex ID

```
1  /**
2  * Calculate the vertex identity.
3  *
4  * Therefore, floor the coordinates to a special number of digits which can be
       set within TOLERANCE_FACTOR constant.
5  */
6  private void calculateId()
7  {
8  // floor the coordinate values to increase the intersection probability
9    double latitude = this.coordinates.getLatitudeDegrees();
10   double longitude = this.coordinates.getLongitudeDegrees();
11
12   double toleranceLatitude =  Math.round(latitude * TOLERANCE_FACTOR) /
       TOLERANCE_FACTOR;
13   double toleranceLongitude =  Math.round(longitude * TOLERANCE_FACTOR)
       / TOLERANCE_FACTOR;
```

```
14
15   this.id = toleranceLatitude + toleranceLongitude;
16 }
```

The probability of connecting trips can be adjusted by setting the constant TOLERANCE_FACTOR. It determines the number of digits behind the comma that will be rounded if it is a set as decimal power (10^n while n is the number of digits behind the comma). During the implementation, n = 1000 (so four digits after comma) was detected as an applicable setting. This setting will connect waypoints of different trips that are within a radius of approximative seven kilometers which can be proved by calculating the distance between the following coordinates using Haversine as shown in listing 5.2.

Listing 5.2: Test method to find a good setting for the intersection probability

```
1 Coordinates one = CoordinateUtil.newCoordinatesBy("55.11111,55.11111");
2 Coordinates two = CoordinateUtil.newCoordinatesBy("55.11111,55.22222");
3
4 System.out.println(CoordinateUtil.calculateDistanceBetween(one, two));
```

The result in listing 5.2.is about seven kilometers. So rounding to four digits results in approximative seven kilometers intersection vertices.

5.5 Changes in modelling

Because Hibernate is used as ORM mapping tool, no changes on the database scheme level itself are required. The property types are modified in the entity classes itself. After a significant change, the Hibernate configuration file must be adjusted so that the new scheme will be created. Since the introduction of a shortest path algorithm, the latitude and longitude data types were changed from float to double to be more precise in terms of the intersection probability to work correctly. Because there's no live environment available, no migration script on database level was required to change the column types since Hibernate can't handle this.

5.6 Shortest Path strategy

The shortest path solution should be easily exchangeable and extendable (functional requirement). Therefore, it was made use of the StrategyPattern to solve the problem of finding the shortest path. As shown in figure 5.2, the interface Routable defines the method *findShortestPath* that each shortest path problem in the car sharing application needs to implement. Per definition, each implementation should solve the Single Pair Shortest Path problem by evaluating both the origin and destination Vertex parameters.

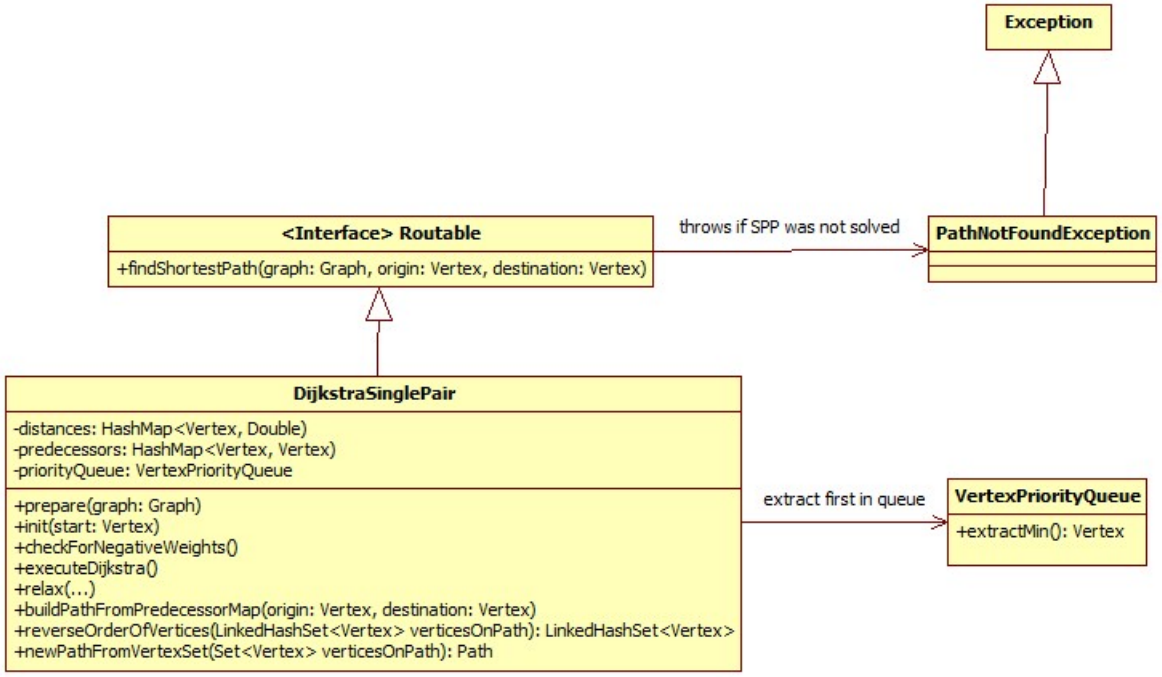

Figure 5.2: UML of the Shortest Path strategy

5.7 Dijkstra implementation

The implementation *DijkstraSinglePair* basically implements Dijkstra by making use of the new Graph classes. It is based on solving the Single Source Path Problem, so internally the algorithm will start at the given origin and find all shortest paths to all other vertices. At the end it will be checked whether the destination vertex is contained in the internal predecessor list to ensure that the Single Pair problem is solved. If not, a *PathNotFoundException* will be thrown. The class contains the following methods:

- *prepare(Graph graph)*: Clear the distances and predecessor maps. The distance map makes it possible to get the marked distance of a vertex. The predecessor map is important for building the Path instance: it is used to build the sequence of vertices from the origin to the destination crossing overlapping trips *after* all shortest paths were determined.
- *init(Vertex start)*: Initialize Dijkstra by setting all distances markers unmarked.
- *checkForNegativeWeights()*: Dijkstra can only solve the shortest path problem for graphs with only positive weights. So this method will

throw an exception if a negative weight was found.

- *executeDijkstra()*: This method works through the priority queue of vertices as long as a vertex with minimum marked distance is contained. It will then call the relax() method for all incident edges of the extracted vertex.
- *relax(Vertex vertexA, Vertex vertexB, double weight, HashMap<Vertex, Double> distances, HashMap<Vertex, Vertex> predecessors)*: This method is the basis of Dijkstra and many other shortest path algorithms. It checks if the the distance to the target vertexB (distance to Vertex A + weight of edge) is less than the currently marked distance. If so, the new distance will be marked using the distance map and the predecessor of vertexB will be set to vertexA which means that a new shortest path was found.
- *buildPathFromPredecessorMap(Vertex origin, Vertex destination)*: This method is called after *executeDijkstra* terminates. It will check whether the destination is contained in the predecessor map and throw a *PathNotFoundException* if not. Otherwise, an instance of *Path* will be built by reversing the order of predecessor vertices, so starting from the origin to the destination vertex.

5.8 Integration into current service

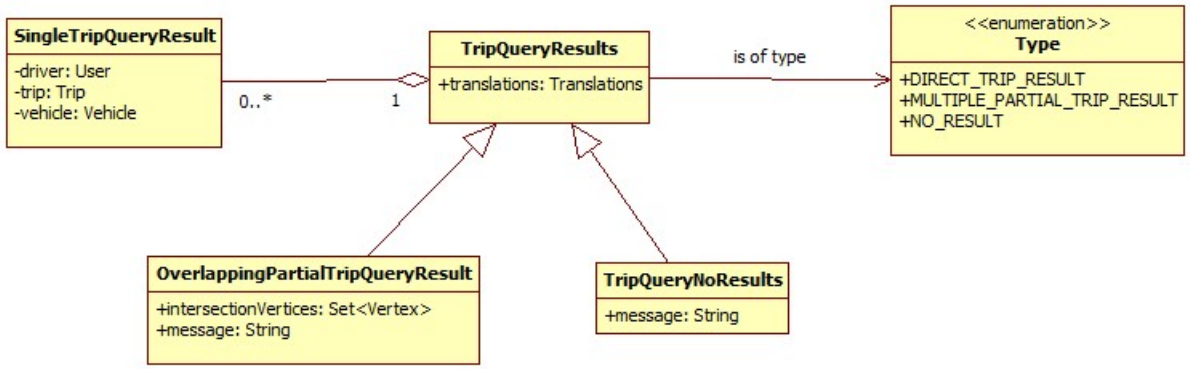

Figure 5.3: UML of the refactored AJAX Query Result models

As stated in chapter 4.2.3, the query result models needed to be replaced by a polymorphistic approach to separate different types of Trip query results, while the existing functionality of finding direct trips should still be

working. So the decision was met to extend the existing class *TripQueryResults* by a new class *OverlappingPartialTripQueryResults* and additionally by TripQueryNoResults to fulfill the process shown in figure 4.1. Figure 5.3 shows the result. An outcome of this solution is that the rendering of found overlapping partial trips in the frontend map immediatly works since the basic JSON data structure coming from the backend didn't change. On the other hand, it allows to add extended behaviour in the frontend such as exposing intersection vertices.

To work with the new structure of query results, the EJB service method to query trips was adjusted as shown in listing 5.3. The method *findOverlappingPartialTrips()* realizes the aimed workflow: first, it triggers the creation of the graph structure and then the Dijkstra Shortest Pair Path Problem.

Listing 5.3: Extended workflow in searchForTrips()

```
1 public TripQueryResults searchForTrips(TripSearchCriteria
      tripSearchModel)
2 {
3   // exclude passenger's own trip first
4   tripSearchModel.setPassenger(this.getCurrentlyLoggedInUser());
5   List<Trip> matchedTrips = this.tripDao.getByCriteria(tripSearchModel)
6
7   // first, filter for direct trips
8   List<Trip> filteredTrips = this.triggerDirectTripsFilteringChain(
      tripSearchModel, matchedTrips);
9
10  if (0 != filteredTrips.size()) {
11    // direct trips were found
12    return this.toQueryResultList(filteredTrips, tripSearchModel);
13  }
14
15  // no direct trips were found, so start to find overlapping partial trips
16  try {
17    Path overlappingPartialTrips = this.findOverlappingPartialTrips(
      tripSearchModel, matchedTrips);
18    return this.toQueryResultList(overlappingPartialTrips,
      tripSearchModel);
19  } catch (PathNotFoundException e) {
20    // no path found at all
21    return this.getNoTripsResult();
22  }
23    }
```

5.9 Frontend adjustments

Since the result JSON of the backend was extended and not basically changed, overlapping partial trips were immediately displayed correctly in the map. It was only changed to evaluate the newly introduced type in TripQueryResult so that specific behaviour can be implemented. A toast message is shown

when a overlapping partial trip result was found instead of a direct trip for example.

5.10 Review

The extension of the car sharing application to also find overlapping partial trips was successful. Now, a multiple trip result will be shown if no direct trip was found for a passenger's search criteria. The new result handling is ready for specific extensions and changes. The results are presented in the next chapter, which also contains an analysis of runtime and complexity.

Chapter 6

Analysis

The car sharing application was extended to also find overlapping partial trips. In this chapter, the new feature will be presented visually. On the other hand, it will be reviewed in terms of runtime and complexity.

6.1 Results

As shown in figure 6.1 and 6.2, the car sharing application will now present a maximum of one overlapping path that crosses partial overlapping trips offered by *different* drivers. The basic interface - a central map element showing the query results if a passengers looks for trips - didn't change at all. If an overlapping trip path was found, the passenger is able to book the partial trips between the intersection points. In figure 6.1, she/he would book a trip from Berlin to Hannover, followed by a trip from Hannover to Cologne. The trips don't need to end in Hannover or Cologne, the same result would also appear if the trips were just passing both cities.

6.2 Usability weaknesses

The new feature comes with usability weaknesses that should be solved in a separate conceptualization:

Exponation of intersections: Intersection locations should be clearly marked. The passenger should be shown a quick-link for example, so she/he is able to identify the overlapping locations of crossing trips quickly.

Travel and intersection times: The current times presented to the passenger are times of whole offered trips. To plan the intersection it is necessary to offer travel times for the partial trips (e.g. from Berlin to Hannover in figure 6.1) only including departure and arrival times.

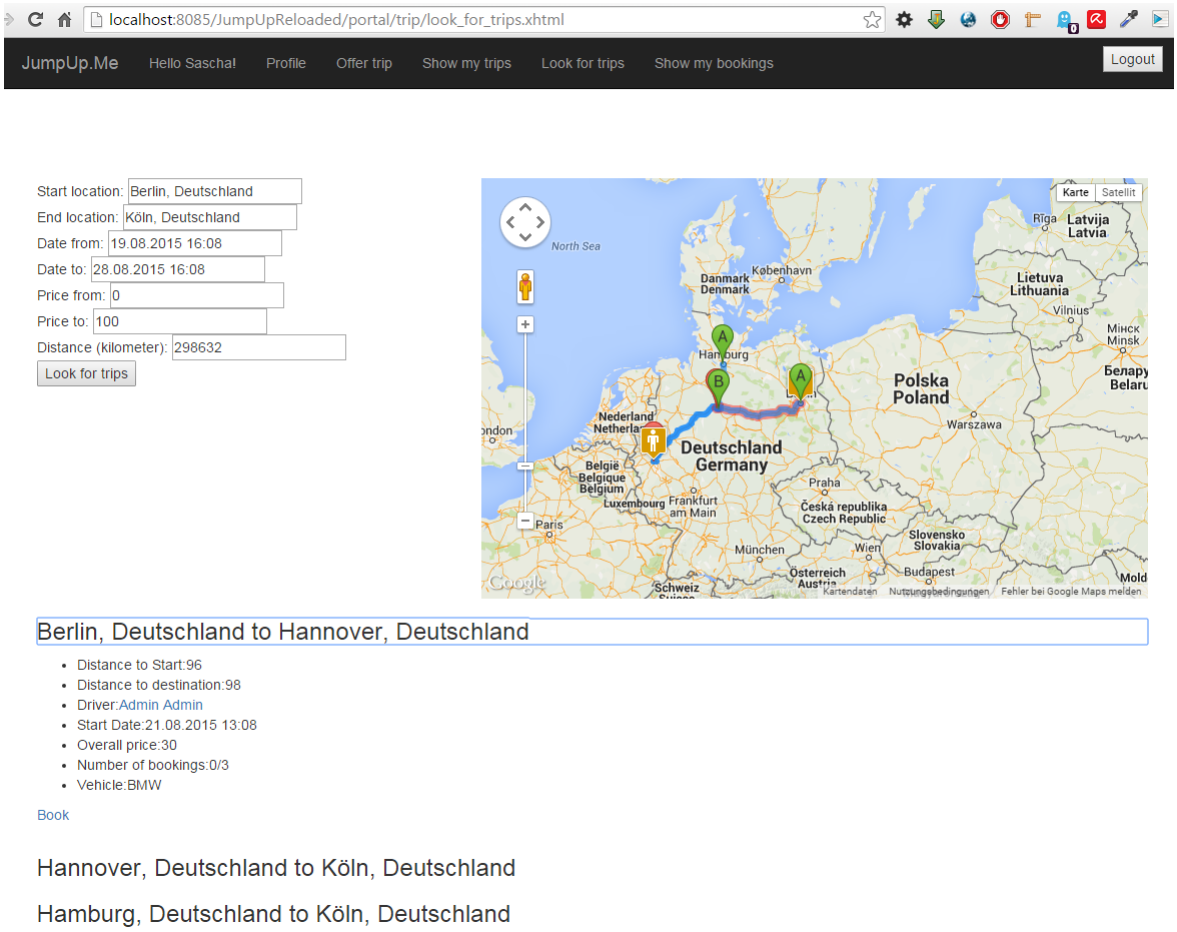

Figure 6.1: Found overlapping trip from Berlin to Cologne with intersection in Hannover - screen 1

Trip filters: The trip HQL queries should be optimized to also evaluate the intersection times so that not realistic paths are filtered out before the shortest path algorithm is executed. For example: The connecting trip from Hannover to Cologne in 6.1 could take place anywhere in between 08/19/2015 and 08/28/2015. The trip results are not filtered to take place on a *single day*.

Offering alternatives: Only one path will be shown currently. As discussed in chapter 3, "K shortest path algorithms" could solve the problem of offering alternatives, for example. In order to get the frontend working, it would be necessary to extend the query result data structure. So instead of returning a list of Trip entities that represent crossing ones on the found shortest path, a list of alternative Path instances should be returned.

For each of the weaknesses, concepts should be found by making use of

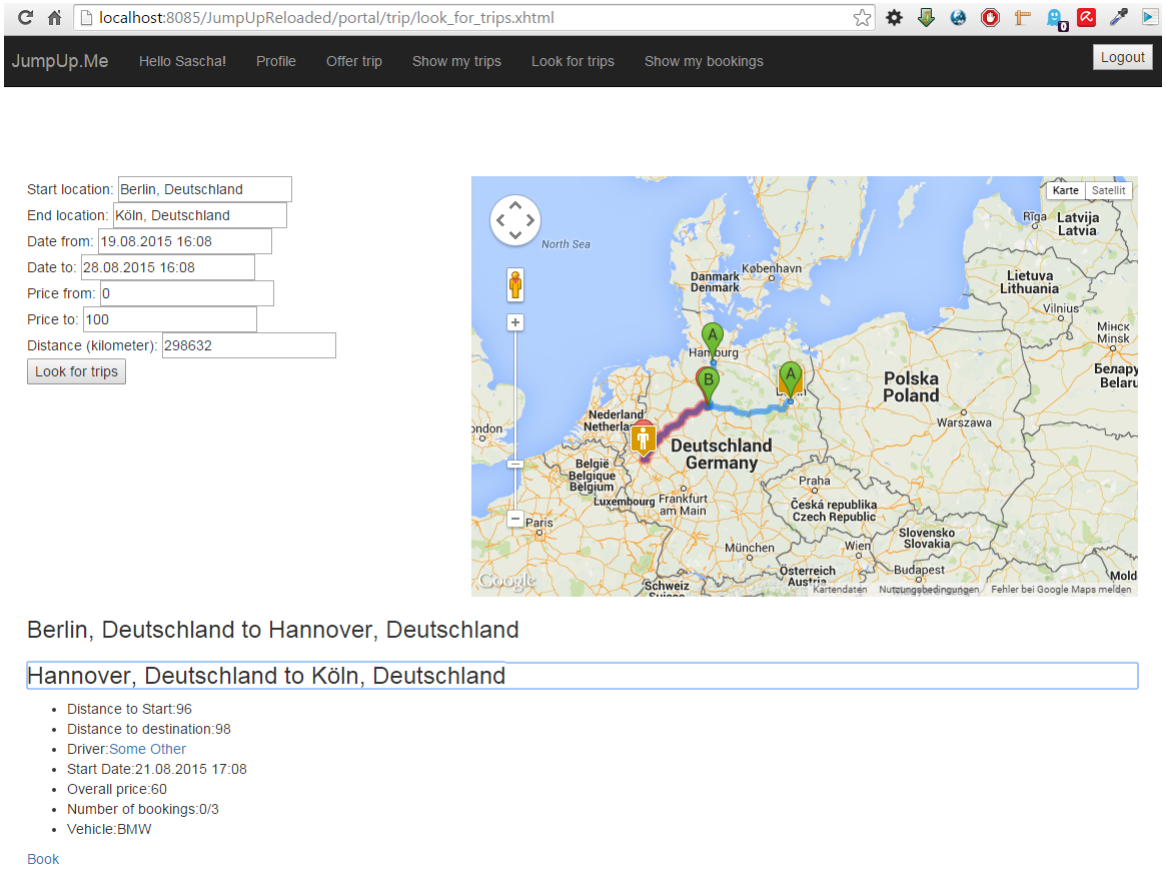

Figure 6.2: Found overlapping trip from Berlin to Cologne with intersection in Hannover - screen 2

usability labs.

6.3 Runtime

The runtime was measured on the environment shown in table 6.1.

Table 6.1: Measurement environment

Processor	Number of cores	Rate (GHZ)	Virtual Memory (GB)
Intel Core I5-3230	4	2.6	12

To measure the runtime, different graph sizes were created. Therefore, trips were created for different users, of different length and number of waypoints. The measurement results are shown in the table below.

Table 6.2: Measurements

Number vertices (N_V)	**Number edges (N_E)**	**Graph Size ($N_V + N_E$)**	**Runtime (ms)**
129	135	264	47
490	523	1013	52
1000	1054	2054	63

6.4 Complexity

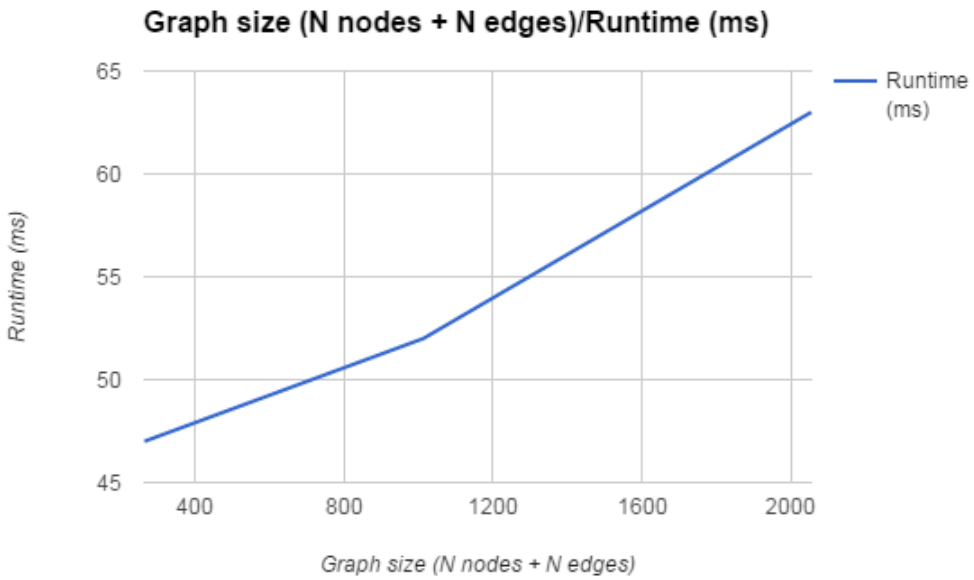

Figure 6.3: Measured runtime as Graph

The results of table 6.2 are visualised in figure 6.3. The problem size is given by the number of vertices plus the number of edges. The curve basically shows a logarithmic growth of the runtime compared to this problem size as it was expected by Dijkstra. Since the problem size is mostly a direct result of the HQL query to filter trips for a special date range, the runtime might be smaller for smaller dateranges.

Chapter 7

Conclusion

As shown in the previous chapter, an applicable extension was implemented. It will be reviewed here. However, the solution is not optimal in terms of usability. Therefore, this chapter also gives an outlook of future extension.

7.1 Review

The shortest path algorithms runs in an acceptable average runtime. The algorithm itself could be optimized by making use of parallel processors if the car sharing application needs to be scaled up in the future. So the chosen algorithm fits perfectly into the existing car sharing application.

The memory usage is quite low which is mostly due to the optimal JEE architecture that was used. During processing, the graph structure will be processed in the EJB server's RAM. The graph structure itself is as compact as possible by making use of adjacency lists and referential dependencies.

The extensions were made by fitting into the current application architecture. The EJB layer allows to reuse the shortest path algorithm everywhere in the application. You could think of introducing a REST service nested in the web layer for example which delegates to the shortest path business logic.

Since design patterns such as the Strategy Pattern and stable models (such as Graph, Vertex, Edge and Path) were used to solve the new task, shortest path solutions can easily be added or extended. The structure is also easily maintainable. The usage of Dijkstra as shortest path algorithm in combination with having a Graph structuring preprocess allows to nest extensions such as offering alternative shortest paths in the graph creation level itself. Only data models that are returned by the AJAX service need to be extended for future extensions in the frontend so the newly introduced shortest path logic itself is stable. A backend developer doesn't need to understand the underlying shortest path logic while extending the service.

7.2 Outlook

Based on the new shortest path algorithm, the frontend application should be extended to offer alternative paths. Therefore, a K-Shortest path approach could be used to randomly delete vertices in the created graph of Trip instances (see 5.3) to get multiple (randomely manipulated) graph structures. The set of shortest path for each of those graph instance would be the alternative multi-stop trips that could be presented to the passenger. This wasn't realized since one of the targets of this work was to evaluate the general shortest path problem in the car sharing application first before the *extended logic* itself is extended.

The frontend should be extended to clearly show intersection locations and times. Thereby, the backend preprocessing logic should be extended to find overlapping trips by evaluating departure and arrival times so that optimal intersections can be found. In general, those extensions should be done within the Usability Engineering discipline.

On the technical base, a better storage concept for the graph structure might get necessary to reduce the runtime of the query trips algorithm. You could think of creating and persisting the Graph structure for a given date range only once instead of creating it during the query of the passenger. So the Graph structure could be reused by multiple users, for example.

So by this work, the technical base and a first working draft to offer multiple partial trips was founded. In the near future, usability optimizations should be planned and conceptualized.

Lists

Print

[**Dijkstra**] Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math., pp. 269–271, 1959.

[**Haversine**] Sinnott, R.W.: Virtues of the Haversine in: Sky and Telescope, vol. 68, no. 2, 1984, 159

[**Krumke**] Krumke, Sven Oliver ; Noltemeier, Hartmut: Graphentheoretische Konzepte und Algorithmen. Berlin Heidelberg New York: Springer-Verlag, 2012. -ISBN 978-3-834-82264-2.

[**Lim**] Yongtaek Lim Hyunmyung Kim: A Shortest Path Algorithm for Real Road Network Based On Path OverlapJournal of the Eastern Asia Society for Transportation Studies, Vol. 6, pp. 1426 - 1438, 2005.

[**Smith**] Smith, David E.: History of Mathematics, Vol. 2. New York: Dover Publs, 1925.

[**Zhan**] Zhan, F. Benjamin: Three Fastest Shortest Path Algorithms on Real Road Networks: Data Structures and Procedures. In: Journal of Geographic Information and Decision Analysis, vol. 1, no.1, pp. 70-82, 1997.

[**Zhou**] Zhou, Sun; Hirokazu, Kato; Yoshitsugu Hayashi: A K-Shortest algorithm for transit network based on partial overlap. Nagoya University, 2005.

[**Yen**] Yen J.Y.: Finding the K shortest loopless paths in a network, Management Science, Vol 17, No. 11, 712-716. 1971.

Web

[**GoogleMapLimits**] https://developers.google.com/maps/usagelimits/. Accessed: 07/22/2015.

[**Hibernate**] http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/. Accessed: 08/06/2015.

[**JEE7Tutorial**] http://docs.oracle.com/javaee/7/JEETT.pdf. Accessed: 07/22/2015.

List of Figures

List of Tables

List of abbreviations

APSP	All Pairs Shortest Path Problem - finding shortest paths between all pairs
EL	Expression Language - language that is used in view scripts to execute Java code
EJB	Enterprise Java Bean - mostly classes that contain business logic
GIS	Geographic Information Systems
HQL	Hibernate Query Language
J2EE	Java 2 Platform Enterprise Edition
JNDI	Java Naming and Directory Interface - standard of an object lookup service
JPA	Java Persistence API - standard for entity persistence, e.g. in relational databases
JSF	Java Server Faces- framework for the creation of graphical user interfaces
MVC	Model View Control
ORM	Object Relational Mapping - mapping between object-oriented and relational databases schemes
SPP	Single Pair Shortest Path Problem - finding shortest paths from one point to all other
SSP	Single Source Shortest Path Problem - finding shortest paths from one point to another