Fraunhofer FOKUS

Kaiserin-Augusta-Allee 31

10589 Berlin

Tel: +49 30 3463-7000

info@fokus.fraunhofer.de

OpenRide Routenmatching

Diese Dokumentation gibt einen Überblick über das Paket ‚de.fhg.fokus.openride.matching', welches die Routenmatching Funktionalität implementiert und nach außen über die Java Bean de.fhg.fokus.openride.matching .RouteMatchingBean' zur Verfügung stellt. Zum Einstieg folgt eine Grobe Skizze des Ablaufes zum Auswerten von Anfragen. Eine Anfrage für Matches hat als Eingabeparameter eine Fahrt bzw. eine Mitfahrt und liefert eine nach Score sortierte Liste von Matches als Ergebnis. Die Auswertung kann wie in Abbildung 1 in sechs Teilschritte zerlegt werden. Die Funktionalität dieser Teilschritte ist auf mehrere Klassen aufgeteilt welche alle von Klasse ‚RouteMatchingBean' konsumiert wird. Diese Teilschritte werden im Verlauf dieser Dokumentation genauer betrachtet.

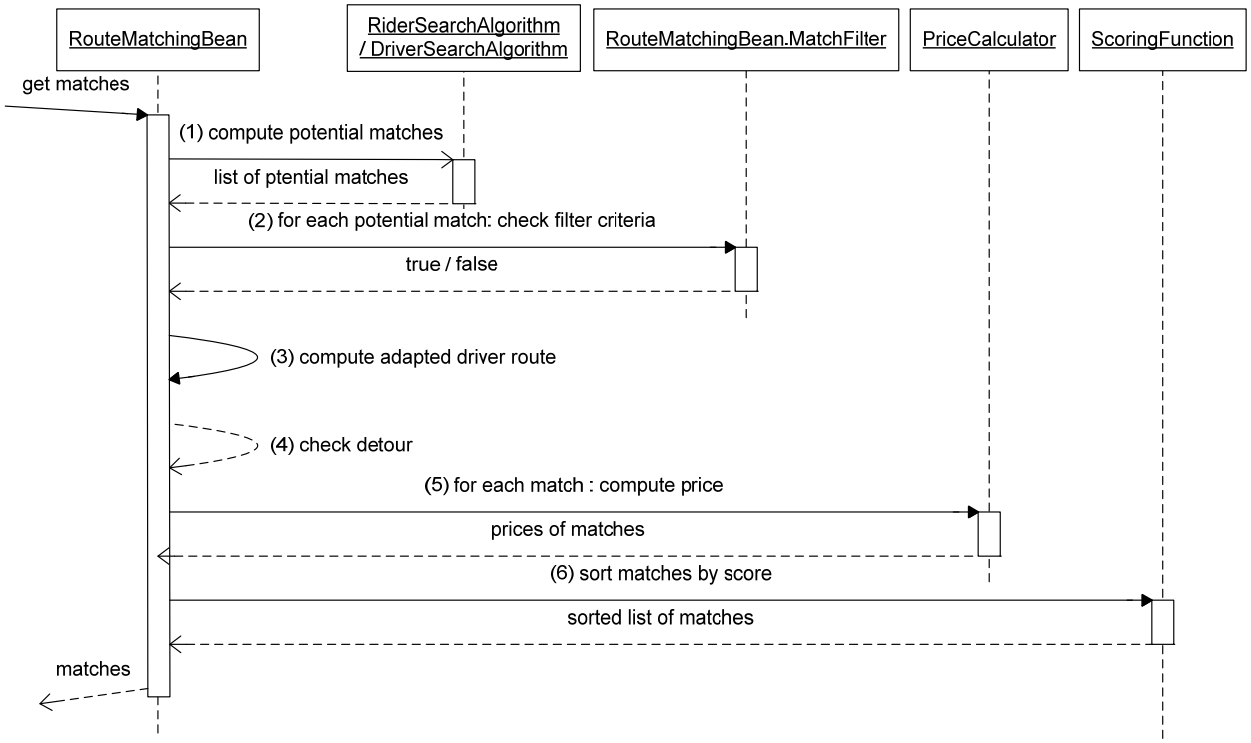


Abbildung 1 Der Prozess beginnt mit einer Anfrage an der Bean Komponente RouteMatchingBean. (1) Zuerst wird mit Hilfe des Circle Overlay Algorithmus eine Menge von potentiellen Matches bestimmt. Es werden genau die Matches bestimmt, welche die Anfragebedingung bezüglich Zeitpunkt und geographischer Position erfüllen. (2) Die zuvor berechneten potentiellen Matches erfüllen eventuell nicht alle Bedingungen der Anfrage. In diesem Schritte werden weitere (kostengünstig validierbare) Kriterien überprüft. Zurzeit sind dies Geschlecht und (Nicht-)Raucher. (3)Für alle verbleibenden Matches wird jetzt eine angepasste Route für den Fahrer berechnet. Diese angepasste Route fährt alle Start- und Zielpunkt aller Mitfahrer an und hat minimale Kosten. Erst nach diesem Schritt ist der tatsächliche Umweg für den Fahrer bekannt. Es kann auch passieren, dass eine solche Route nicht existiert, und das Match verworfen werden muss. (4) Matches deren tatsächlicher Umweg größer ist als der vom Fahrer angegebene maximale Umweg werden ebenfalls verworfen. (5) Für alle verbleibenden Matches wird der Preis berechnet. Zurzeit basiert die Preisberechnung auf dem Umweg für den Fahrer, sowie auf der Distanz der Mitnahmestrecke. (6) Alle verbliebenen Matches werden mit einer ‚Scoring-FunKtion' bewertet und diesbezüglich absteigend sortiert.

Circle Overlay Algorithm

Diese Heuristik berechnet Matches ausschließlich bezüglich geographischer Position und Zeit. Es gibt zweierlei Typen von Anfragen, welche auf unterschiedliche Art behandelt werden. Zum einen muss für einen bestimmten Fahrer eine Menge potentieller Mitfahrer gefunden werden, zum anderen wird für einen speziellen Mitfahrer eine Menge von potentiellen Fahrern ermittelt. Diese beiden Heuristiken sind in den Klassen ‚RiderSearchAlgorithm, sowie, DriverSearchAlgorithm, implementiert und werden grob skizziert. Für alle Schritte wird immer auch die Zeit überprüft, was im Folgenden nicht mehr extra erwähnt wird.

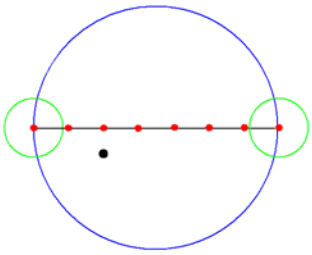
Finde Mitfahrer für Fahrer

Die Route des Fahrers wird durch eine rel. kleine Anzahl äquidistanter Routenpunkte angenähert. Zum finden potentieller Mitfahrer wird jetzt in mehreren Schritten die potentielle Ergebnissemenge durch Ausschlusskriterien verkleinert. Der Algorithmus ist variabel bezüglich des Abstandes der

äquidistanten Routenpunkte sowie bezüglich des Radius der Kreise für die Überdeckung. Diese können als Tuning-Parameter übergeben werden, wobei eine Abhängigkeit zwischen diesen Parametern besteht welche beachtet werden sollte (siehe Ergebnisse der Bachelor Arbeit von Martin). Implementiert in RiderSearchAlgorithm.findRiders(…).

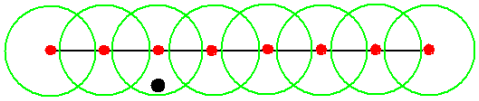
Schritt 1

Überdeckung der Kreise um Start und Ziel (grün) sowie um den Mittelpunkt der Route (blau)mit allen Einstiegspunkten aller Mitfahrer. Mitfahrer mit nicht überdeckten Einstiegspunkten werden ausgeschlossen.



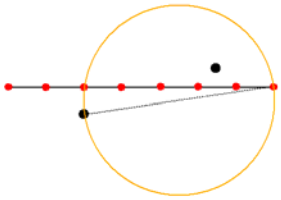
Schritt 2

Überdeckung der Kreise um alle Routenpunkte mit allen Einstiegspunkten der potentiellen Mitfahrer (stärkere Eingrenzung als in Schritt 1). Mitfahrer mit nicht überdeckten Einstiegspunkten werden ausgeschlossen.



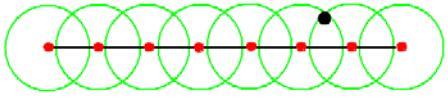
Schritt 3

Für jeden potentiellen Mitfahrer : Überdeckung des Ausstiegspunktes mit einem Kreis durch den Mittelpunkt von Einstiegspunkt und Ziel des Fahrers. Mitfahrer mit nicht überdeckten Ausstiegspunkten werden ausgeschlossen.



Schritt 4

Für jeden potentiellen Mitfahrer : Überdeckung des Ausstiegspunktes mit den Kreisen um die Routenpunkte (stärkere Eingrenzung als in schritt 3). Mitfahrer mit nicht überdeckten Ausstiegspunkten werden ausgeschlossen.



Finde Fahrer für Mitfahrer – DriverSearchAlgorithm.findDrivers(…)

Diese Heuristik ist relativ trivial. Berechne zwei Kreisüberdeckungen für die Kreise um Einstiegs- und Ausstiegspunkt mit den Routenpunkten. Alle Routen welche mit beiden Kreisen überlappen sind Teil des Ergebnisses.

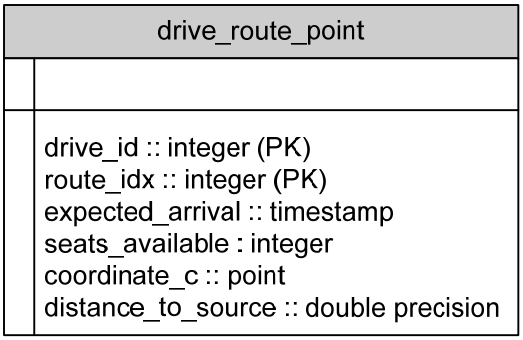
Implementierung

Die Auswertung der Anfrage Erfolge auf einer Postgresql Datenbank mit Postgis Erweiterung. Alle Fahrten und Mitfahrten werden in der Datenbank gehalten wobei die Geographischen Informationen mittels des geographischen Datentyps ‚Point' aus der Postgis Erweiterung repräsentiert werden. Um den Circle Overlay Algorithmus verwenden zu können muss das Schema eine Supermenge folgende r Relationen beinhalten.

Um die Anfrage relativ effizient implementieren zu können wird eine temporäre Tabelle verwendet um das Ergebnis der ersten Unteranfrage zu halten. Dadurch konnte die darauffolgende Anfrage zum auswählen der Ergebnismenge effizienter ausgeführt werden.

Datenbank Schema

(1) Für eine Fahrt existiert für jeden Routenpunkt ein Eintrag in dieser Tabelle. Diese Tabelle enthält die äquidistanten Routenpunkte aller Fahrtangebote. Die Tabelleneinträge werden von der Bean Komponente DriverUndertakesRideController angelegt / verwaltet.



`drive_id:` identifiziert das Fahrtangebot.

`route_idx :` Index des Routenpunktes bezüglich der Route, aufsteigend von Start nach Ziel vergeben.

`expected_arrival :` Geschätzte Passierzeit des Fahrers für diesen Routenpunkt.

`seats_available :` Anzahl der Sitzplätze die zwischen diesem und dem folgenden Routenpunkt frei sind.

`coordinate_c :` karthesische Koordinate des Routenpunktes (WGS86 auf Ebene projiziert)

`distance to source :` Zurückgelegte Strecke des Fahrers vom Startpunkt bis zu diesem Routenpunkt.

(2) In dieser Tabelle existiert genau eine Zeile für jede Mitfahrt. Die Tabelleneinträge werden von der Bean Komponente RiderUndertakesRideController angelegt / verwaltet.

riderundertakesride
riderroute_id :: integer (PK) start_time_earliest :: timestamp start_time_latest :: timestamp startpt_c :: point endpt_c :: point

`riderroute_id` : Identifiziert die Mitfahrt.

`start_time_earliest` : Frühest möglicher Abholzeitpunkt.

`start_time_latest`: Spätest möglicher Abholzeitpunkt.

`startpt_c` : Karthesische Koordinate des Abholpunkte.

`endpt_c` : Karthesische Koordinate des Ausstiegspunktes.

Parameter

Die Parameter sind im Code der Klassen ‚RiderSearchAlgorithm, und ‚DriverSearchAlgorithm, dokumentiert. Variabilität besteht wie bereits erwähnt bezüglich des Abstandes der äquidistanten Routenpunke sowie bezüglich der Kreisradien. Untersuchungen dazu sind in Martin's Bachelor Arbeit zu finden. Die beiden o.g. Klassen werden ausschließlich von der Klassen ‚RouteMatchingBean' verwendet. Innerhalb dieser Klasse werden diese variablen Parameter konfiguriert.

Der derzeitige Stand erlaubt keine Änderung dieser Parameter im laufenden Betrieb. D.h. bei Änderung des Abstands der Routenpunkte bzw. der Radien der Kreise müssen für alle bereits existierenden, noch gültigen Fahrtangebote die äquidistanten Routenpunkte neu berechnet werden. Würde man sich zusätzlich für jede Fahrt den bei der Suche zu verwendenden Kreisradius speichern, könnte man eine Änderung der Parameter ermöglichen ohne eine Neuberechnung der Routenpunkte erforderlich zu machen. Das Circle Overlay für bereits existierende Angebote würde dann weiterhin mit den alten Parametern berechnet, für neue hinzukommende Angebote würden die neuen Parameter verwendet.

Rückgabewert des Circle Overlay Algorithmus

Die Antwort auf Anfragen enthält eine Liste von Instanzen der Klasse ‚PotentialMatch'. Ein Potential Match repräsentiert ein Paar bestehend aus Fahrt und Mitfahrt welches die die Anfragebedingung erfüllt (bezüglich geographischer Position und Zeit). Bei entsprechender Parameterwahl werden alle solche übereinstimmenden Paare gefunden und zurückgegeben. Die Reihenfolge der Listeneinträge ist nicht festgelegt.

Filterung von potentiellen Matches

Ein rechenintensiver Schritt potentielle Matches auf Gültigkeit zu überprüfen ist die Überprüfung auf den Umweg. Dies ist deshalb aufwendig, da hierfür eine neue Route für den Fahrer gefunden werden muss. Deshalb wird die Menge der potentiellen Matches zuerst durch einfachere Filterkriterien ausgedünnt. Dies passiert in der inneren Klasse ‚RouteMatchingBean. MatchFilter. Hier werden folgende Bedingungen u.a. gemäß den Präferenzen des Fahrers / des Mitfahreres überprüft. Es folgt eine Liste der zu überprüfenden Bedingung ggf. mit den jeweiligen Konfigurationsmöglichkeiten:

Triviale Bedingungen

1. Geschlecht :
    - Bisher einzig möglich Restriktion : ‚Fahrt von Frauen für Frauen'
2. Rauchen / Nichtrauchen :
    - Egal
    - Erwünscht (es muss erlaubt sein zu rauchen)
    - Nicht erwünscht (es darf nicht erlaubt sein zu rauchen)
3. Freie Sitzplätze :
    - Überprüft ob genügen freie Sitzplätze vorhanden sind um alle Personen eines Mitfahrgesuchs mitnehmen zu können (ein Mitfahrangebot kann mehrere Mitfahrer beinhalten).
4. Ein Nutzer kann niemals mit sich selbst matchen (zum Testen evtl. sinnvoll.)
5. Bereits gebuchte Mitfahrten müssen ignoriert werden.

Die Überprüfung jeder einzelnen dieser Bedingungen kann mithilfe der statischen konfigurations Variablen der Klasse ‚RouteMatchingBean' an- / aus geschaltet werden.

Umwegbereitschaft

Um den Umweg zu Berechnen wird ein Umplanen der Route des Fahrers notwendig (Details dazu, siehe Abschnitt ‚Umplanen der Fahrerroute'). Aus der Differenz der Längen der alten - und neuen Route ergibt sich der Umweg. Es kann jetzt die Bedingung Umwegbereitschaft des Fahrers überprüft.

Alle verbleibenden potentiellen Matches werden vom Algorithmus zurückgegeben und vorher um Zusatzinformationen erweitert.

Fahrtpreisberechnung

Für alle nach der Filterung verbleibenden Matches wird der Fahrtpreis berechnet. Dafür existiert die abstrakte Klasse ‚PriceCalculator'. Sie enthält eine abstrakte Methode welche den Fahrtpreis in Cents in Abhängigkeit von Umweg und Mitnahmestrecke berechnet. Über die Methode getInstance() wird eine Instanz der Klasse ‚PriceCalculator' zurückgeliefert, welche das aktuell zu verwendende Preismodell implementiert. Bisher sind zwei unterschiedliche Preismodelle verschachtelt innerhalb der Klasse ‚PriceCalculator' implementiert.

1. Abgestuftes Preismodell (siehe PriceCalculator .STEPPED_PRICE_CALCULATOR)
    - Der Preis pro Kilometer ist Abhängig von der Länge der Mitnahmestrecke
    - Hinzu kommt eine Kilometerpauschale für den Umweg der anfällt um den Mitfahrer abzuholen.
    - Auf folgende Preismatrix wurde sich geeinigt

Streckenabschnitt (Km von - bis)	Preis (Cents)
0 - 10	20
10 - 50	10
50 - …	5

    - Preis pro Umweg-Kilometer : 35 Cents
2. Lineares Preismodell (siehe PriceCalculator .LINEAR_PRICE_CALCULATOR)
    - Fester Preis pro Kilometer : 25 Cents
    - minimaler Preis : 50 Cents
    - maximaler Preis : 300 Cents

- Anzurechnende Kilometer ergeben sich aus Mitnahmedistanz plus Umweg

Ranking

Alle Matches wurden jetzt anhand von Ausschlusskriterien gefiltert und um Zusatzinformation (Preis, Mitnahmestrecke, Umweg) erweitert. Vor dem Ranking werden alle Matches als Instanz der Klasse MatchEntity repräsentiert. Um eine Priorisierung dieser Matches für den Buchungsprozess festzulegen wird ein Ranking der Matches vorgenommen werden. Für jedes Match wird ein sog. Score vergeben. Anschließende werden alle Matches absteigend nach ihrem Score sortiert.

Es folgt eine kurze Erfahrung bei der Implementierung : Ein erster Ansatz hatte versucht die Sortierung durch Implementierung des ‚Kleiner-Gleich-Operators' zusammen mit Quicksort zu realisieren. Dabei sollten das Ranking jeweils unterschiedlich aus Sicht des Mitfahrers bzw. aus Sicht des Fahrers stattfinden und mehrere Kriterien berücksichtigen. Durch die unterschiedliche Behandlung sowie das Einbeziehen mehrerer Kriterien wurde nichtmehr leicht ersichtlich ob der implementierte Vergleichsoperator transitiv ist. Dies ist jedoch eine Voraussetzung dafür, das vergleichsbasiertes sortieren überhaupt funktioniert. Eine Implementierung über einen Score impliziert dies direkt. Nach einigen Überlegungen konnte kein Argument dafür gefunden werden, ein Ranking für Fahrer bzw. Mitfahrerseite unterschiedlich vorzunehmen.

Die Bewertungsfunktion ist analog zur Preisberechnung als abstrakte Klasse namens ‚ScoringFunction' umgesetzt. Diese bekommt eine Instanz der Klasse ‚MatchEntity' und liefert einen double Wert als Score, wobei größere Werte eine höhere Priorisierung darstellen. Es gibt bisher nur eine naive Implementierung dieser Klasse (siehe ScoringFunction .SIMPLE_SCORING_FUNCTION). Es wird eine Bewertung in Abhängigkeit von Mitnahmestrecke und Umweg berechnet, wobei diese beiden Faktoren mit einem festgelegten Faktor gewichtet in den resultierenden Score-Wert eingehen.

Rückgabewert der Komponente ‚RouteMatchingBean'

Eine Liste mit Matches absteigend sortiert nach dem score Wert. Jedes Match ist als Instanz der Klasse ‚MatchEntity' repräsentiert. Diese Klasse wird schließlich vom Buchungsprozess weiterverwendet um zum Einen den jeweils aktuellen Zustand des Buchungsautomaten zum Anderen eine Warteschlange mit schon berechneten aber vom Buchungsprozess noch nicht betrachteten Matches zu speichern.

Umplanen der Fahrerroute

Bereits beim einstellen des Fahrtangebots hat der Fahrer einen Start- und Zielpunkt der Fahrt und eventuell weitere Orientierungspunkte (um die Route an seine eigenen Bedürfnisse anzupassen) festgelegt. All diese Punkte müssen in jedem Fall auch von der zu berechnenden Route angefahren werden. Es kann dabei passieren, dass die vom Fahrer festgelegten Orientierungspunkte zusammen mit den neu hinzukommenden Mitnahme und Absetzpunkten zu nicht ganz optimalen Routen führt.

Damit ist gemeint dass der Fahrer die Orientierungspunkte mit dem Wissen um die Mitnahme- und Absetzpunkte seiner Mitfahrer anders festgelegt hätte. Diese waren allerdings zum Zeitpunkt des Einstellens des Fahrtangebots noch nicht bekannt. Andererseits können die Orientierungspunkte nicht einfach vernachlässigt werden, da sonst die Anpassung der Route, welche vom Fahrer durch die Orientierungspunkte vorgenommen wurde, verloren gehen. Dies lässt sich am Fall des perfekt-Match verdeutlichen, d.h. ein Mitfahrer kann über die komplette Distanz mitgenommen werden. Werden nun die Orientierungspunkte beim Neuplanen der Route weggelassen, ergibt sich die ursprünglich Route, welche nicht den Wünschen des Fahrers entsprach. Es wird vermutet dass das Beibehalten der Orientierungspunkte zu brauchbaren Ergebnissen führt.

Die Problemstellung für das Neuplanen der Route, d.h. eine Neuberechnung der Route des Fahrers unter Einbezug des Start- und Zielpunktes einer bestimmten Mitfahrt lässt sich wie folgt formulieren:

Gegeben ist eine Menge von Punkten P die auf jeden Fall angefahren werden müssen. Diese beinhaltet die Fahrtpunkte (Startpunkt, Zielpunkt, Orientierungspunkte) und die Start- und Zielpunkte aller Mitfahrer. Gesucht ist eine Reihenfolge zum Anfahren der Punkte, sodass die Gesamtroute eine minimale Länge hat.

Lösungsansatz 1

Es ist nicht jede der |P | ! denkbaren Reihenfolge des Anfahrens der Punkte möglich :

- Startpunkt und Zielpunkt sind immer zuerst / zuletzt
- Ein Mitnahmepunkt eines Mitfahrers muss immer vor seinem jeweiligen Absetztpunkt angefahren werden
- Die Ordnung innerhalb der Orientierungspunkte bleibt erhalten.

Für alle gemäß obigen Regeln möglichen Reihenfolgen von Punkten :

- Berechne die Länge der Route (erfordert das Berechnen von |P| - 1 Teilrouten)
- Wähle die Route mit minimaler Gesamtlänge

Lösungsansatz 2 (implementiert)

Die Anzahl der (Teil-)Routenberechnung lässt sich weiter verringern indem die Ergebnisse vorheriger Neuberechnungen der Route zu derselben Fahrt wiederverwendet werden. Gegeben eine Liste P = (P[0], P[1], … P[n-1])von Punkten die aktuell von der Route angefahren werden müssen. Die Punkte sind nach der Reihenfolge in der Sie angefahren werden sortiert. Neu hinzu kommen jetzt je ein Mitnahme und ein Absetzpunkt.

Das Problem kann jetzt mit folgendem Algorithmus gelöst werden :

1. Finde Position in der Liste, an welcher der Mitnahmepunkt eingefügt werden soll.
    a. Der Punkt wird an Position i eingefügt wenn die Summe der Längen der beiden Routen P[i-1] -> P[i] und P[i] -> P[i+1] minimal ist.
    b. Mit $0 < i < n-1$
2. Füge Mitnahmepunkt an der in a. berechneten optimalen Position s ein.
3. Finde Position in der Liste, an welcher der Absetzpunkt eingefügt werden soll
    a. Analog zu 1.a.
    b. Mit $s < i < n-1$

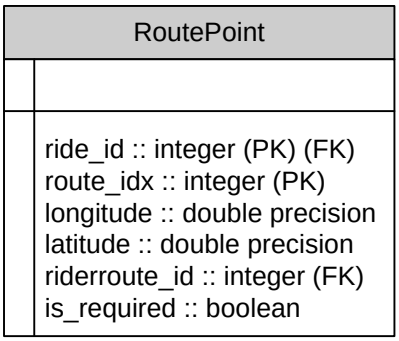
Dieser Ansatz ist wesentlich effizienter als der 1. Ansatz. Im erwarteten Fall müssen insgesamt 3 * |P| Routenberechnungen ausgeführt werden. Eine weitere Erhöhung der Effizienz der Berechnung ist aus meiner Sicht nicht ohne größeren Aufwand machbar.

Implementierung

Das Neuberechnen der Route ist innerhalb der Methode ‚RouteMatchingBean.computeAdaptedRoute' wie oben beschrieben implementiert. Diese Methode wird zur Berechnung des Umwegs für jedes Match 1x aufgerufen. Wird eine Fahrtbuchung an der Komponente DriverUndertakesRideController realisiert, so wird diese Methode ebenfalls verwendet, um die neue Route des Fahrers zu erhalten und zu persistieren. Alle Routenpunkte der aktuellen Route einer Fahrt (mit allen Wegpunkten) werden dafür in einer separaten Tabelle in der Datenbank gehalten. Jeder dieser Routenpunkte hat dabei ein Flag, welches gesetzt ist falls der Routenpunkt zwingend angefahren werden muss. Beim neuberechnen der Route werden alle Routenpunkte mit gesetztem Flag aus der Datenbank gelesen, welche nach der Reihenfolge des Anfahrens sortiert werden können. Folgend werden die Einfügepositionen des Mitnahme und Absetztpunktes wie in Ansatz 2 beschrieben ermittelt.

Datenbank Schema

Die Tabelle zum Speichern der Routenpunkte. Diese enthält alle im Kartenmaterial enthaltenen Koordinaten der Route und wird nicht von dem Circle Overlay Algorithmus verwendet. Die hier gespeicherten Routenpunkte dienen zum einen als Information für die Neuberechnung von Routen, zum anderen können die Punkte zum Anzeigen der Route innerhalb der GUI des Fahrers verwendet werden. Lässt sich ein Fahrer seine aktuelle Route anzeigen wird so keine Berechnung der Route nötig. In diesem Fall wird die Route einfach aus der Datenbank geholt.



`ride_id` : Identifiziert Fahrt.

`route_idx` : Index innerhalb der Route, absteigen von Startpunkt zum Zielpunkt.

`longitude` : geographische Position

`latitude` : geographische Position

`riderroute_id` : Identifiziert Mitfahrt, falls der Punkt ein Mitnahme oder Absetzpunkt einer Fahrt ist

`is_required` : True falls der Punkt zwingend angefahren werden muss.(Für die Neuplanung der Route)

Limitierung der Orientierungspunkte des Fahrers

Die Anzahl der Routenberechnungen pro berechnetem Match steigt linear mit der Anzahl der vom Fahrer angegebenen Orientierungspunkte. Deshalb muss die Anzahl der Orientierungspunkte auch zum Schutz vor Angriffen und hinsichtlich Performanz limitiert werden. Da die Routenplanungskomponente kurze Antwortzeiten bietet ist dies Performancetechnisch vermutlich zuerst einmal unkritisch, sollte aber im Auge behalten werden.

Weitere Optimierung

Sollte sich die Umwegberechnung irgendwann zu einem Flaschenhals entwickeln, sind weitere vielversprechende Optimierungen denkbar. Eine Möglichkeit besteht darin, den Highway Hierarchies Algorithmus um eine für diesen Fall angepasste Anfrage zu erweitern. Der entstehende Anfragealgorithmus ließe sich vermutlich aus dem normalen HH Anfrage Algorithmus und dessen Many-To-Many-Shortest-Path Variante ableiten.

Dead Code

Die Klassen ‚Match, und ‚Constants' werden nicht benötigt und können entfernt werden. Dies konnte nicht gemacht werden, da weitere dead Code von diesen Klassen Abhängig ist.

Routenplanung

Die Planung von Routen erfolgt in einem separaten Paket : , de.fhg.fokus.openride.routing' Die funktionalität zur Routenplanung wird über die Schnittstelle Router mittels der Komponete ‚RouterBean' bereitgestellt. Hier gab es zuerst eine Implementierung auf Basis von PgRouting. Die Klasse ‚osm.OsmRouter' dafür existiert noch, wird aber nicht mehr verwendet. Startdessen wird sie durch die Klasse ‚osm.OsmHHRouter' ersetzt. Die Routenplanung erfolgt jetzt auf Basis der mapsforge Bibliothek.

Insgesamt ist die Klassenstruktur innerhalb des Pakets unnötig aufgebläht und sollte nach dem ‚Keep It Simple' Prinzip überarbeitet werden. Auch Methoden zur Interpolation der Route sollte aus dem Router Interface herausgezogen und in das RoutenMatching Paket verlagert werden.