Route Matching in OpenRideShare

Jochen Laser

April 6, 2015

Abstract

This document describes the Circle Overlay Algorithm used to match offers and requests to each other in Open Ride Share.

Contents

1 Introduction 4

1.1 Rideshareing and Dynamic Rideshareing 4

1.2 Route Matching 5

1.3 Offers, Requests and Route Matching in more Detail 5

1.4 Creating an Offer 5

1.5 The Route 6

1.6 Creating a Request 6

1.7 Route Matching in more Detail 7

1.7.1 Finding matching Requests for a given Offer 7

1.7.2 Finding matching Offers for a given Request 8

2 Searching for Riders 10

2.1 Step 1: Create Route and Routepoints 10

2.2 Step 2: Create Drive Route Points 11

2.2.1 The Circle Overlay Algorithm 11

2.2.2 Drive Route Points 12

2.2.3 Parameters for the Circle Overlay Algorithm 12

2.3 Step 3: Scanning the Corridor 13

2.4 Step 4: Filtering by simple Criteria 13

2.5 Step 5: Calculating the Detour 13

2.6 Step 6: Applying the Sorting Function 13

2.7 Step 7: Applying Limits 13

3 Searching for Drivers 14

3.1 Step 1: Preselecting Offers 14

3.2 Step 2: Filtering by simple Criteria 15

3.3 Step 3: Calculating the Detour 15

3.4 Step 4: Applying the Sorting Function 15

3.5 Step 5: Applying Limits 15

4 Choosing the Test Radius around a DRP **16**

4.1 Deflection from the direct Line 16

4.1.1 Maximum Deflection appears at Paths with *exactly one* route point 16

4.1.2 The Ellipse 17

4.1.3 The Maximum Deflection as Function of the direct Distance between Drive_Route_Points 18

4.2 The Testradius 19

5 Choosing the Distance between DRPs **21**

5.0.1 The Defect 21

5.1 Choosing d for a straight Path 22

5.1.1 Optimal Choice of d for Straight Line and Testradius t_{r3} 23

Index **25**

Chapter 1

Introduction

This document discusses *routeMatching* in the *OpenRideShare* Application.

OpenRideShare (henceforth abbreviated *"ORS"*) is an application that provides *RideSharingServices.* The OpenRideShare Software is largely based on the codebase a predecessor named "OpenRide". OpenRide was provided by the *FOKUS, Fraunhofer Institute für offene Kommunikationssysteme.* OpenRide went OpenSource in 2011, but was not developed further by FOKUS. So, the code was picked up by a bunch of enthusiastic enthusiasts and developed further, now under the label *"OpenRideShare".*

1.1 Rideshareing and Dynamic Rideshareing

Suppose that Janet wants to drive from Douglas to Ramsay (both towns on the Isle of Man in the Irish Sea), and that Janet has some empty seats in her car, which she would like to share with someone else travelling that direction. Next, suppose that at the same time Bob wants to travel from Baldrine (which is close to Douglas) to Port Lewaigue (which is close tho Ramsay). Picking up Bob at Baldrine and dropping him at Port Lewaigue would add only some small detour to Janet's route. If this detour is within a maximum detour that Janet is willing to go, than it makes sense for Janet and Bob to travel together, i.e share a ride, and maybe a portion of the ride's costs.

OpenRideShare offers the capabilitiy to bring drivers (like Janet) and riders (like Bob) together. Via a mobile interface it is also possible for drivers to find matching requests in "real time", while the driver is already on the road. This is sometimes called *"Dynamic Rideshareing".*

1.2 Route Matching

Route Matching is the act of bringing together matching offers and requests. This is the core functionality of OpenRideShare (and, of course any dynamic ridesharing application) Technically, route matching comprises either one of the following two actions:

1. When a new offer is created, search for matching requests
2. When a new request is created, search for matching offers

As seen in the previous short example with Janet and Bob, the places where Bob has to be picked up and dropped are close to Janet's route, but not exactly on it, which forces Janet to slightly alter her route, and go a small detour. Finding matches thus comprises calculating and testing alternative routes, which is a notoriously costly operation. Thus, the ORS's route matching algoritms aim at doing a preselection of rides that limits the actual cases where an altered route has to be calculated to as few as possible matches. The rest of this document is mainly about how this is done.

1.3 Offers, Requests and Route Matching in more Detail

In this section we discuss the process of creating offers and request, the creation of routes, and route matching from a highlevel perspective.

1.4 Creating an Offer

Before starting her journey, Janet enters some data about the journey into the ORS. These data mainly comprise:

- The place where the journey starts (In our example: *"Douglas"*)
- The place where the journey ends (In our example: *"Ramsay"*)
- Optionally some additional points she wants to visit on her way (called *"waypoints"*). For simplicity, we do not have waypoints in our example.
- Date and time when Janet wants to start
- A *"maximum detour"* parameter, giving the maximal detour that Janet is willing to go when picking up coriders.

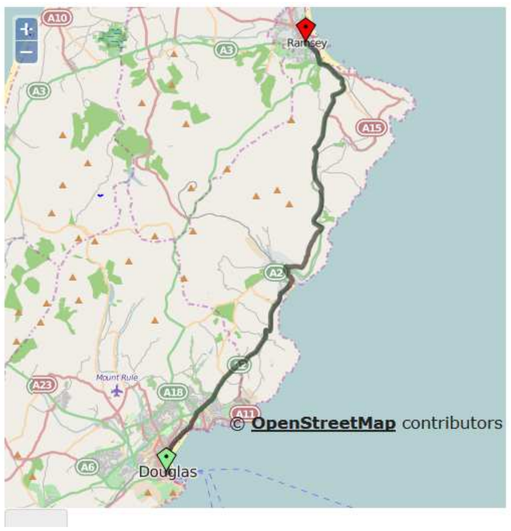

Figure 1.1: Janet's route from Douglas to Ramsay as calculated by ORS

- The Number of free seats Janet wants to offer
- Optionally a number of criteria such as gender, smoker/nonsmoker, etc which can be used to filter potential riders

1.5 The Route

Given these data, ORS invokes a *Routing engine* to calculate a *route* for Janet. Technically, the route is an ordered list of so called *Routepoints*, with each Routepoint containing the following data:

- Spatial Coordinates in polar longitude/latitude form and in addition in some suitable local carthesian coordinate system for speedy calculation of distances)
- A timestamp describing when Janet passes this point
- A reference to Janet's offer

Figure 1.1 shows Janet's route from Douglas to Ramsay, as calculated by ORS.

1.6 Creating a Request

Now Bob wants to travel from Baldrine to Port Lewaigue, which is roughly on Janet's way. Not wanting to drive with his own car, Bob takes his chances with OpenRideShare and creates a *request* in ORS. To create a request, Bob has to enter the following data:

startpt The place where the journey starts (In our example: *"Baldrine"*)

endpt The place where the journey ends (In our example: *Port Lewaigue*)

startTimeEarliest A timestamp describing the earliest time when Bob wants to start.

startTimeLatest A timestamp describing the latest time when Bob wants to start.

criteria Optionally: a number of criteria such as gender, smoker/nonsmoker, etc which can be used to filter potential drivers.

1.7 Route Matching in more Detail

The route matching step is applied immeadiately after creating an offer or request. We discuss route matching for a newly created offer briefly in 1.7.1, and extensively in section 2. Route matching for a newly created request is discussed briefly in 1.7.1, and extensively in section 3.

1.7.1 Finding matching Requests for a given Offer

As seen in section 1.5, along with an offer a route gets created. Next, along that route a corridor of width *detour* is searched for requests that have either start or endpoint in that corridor.

Here *corridor*, means the places that have *direct distance detour* from the route, not *road distance*. While for calculating the road distance expensive routing functionality has to be used, direct distances (aka "as the crow flies") can be calculated with little effort and provide a good upper bound for road distances. If a request is found that has both start and endpoint in that corridor, and with the startpoint matching the temporal bounds, then the associated request is preselected for final filtering.

In the final filtering step, first the "optional parameters" (smoker, gender,...) are tested. For all request passing this filter, finally an exact route containing the start and endpoint of the request is calculated. If this route does not exceed the original route by more then detour, the request is considered to be successfully matched. It then gets displayed to the rider and driver in the frontend for approval.

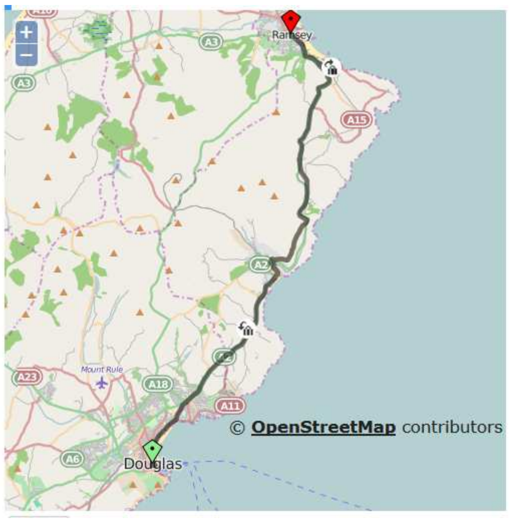

Figure 1.2: Janet's route from Douglas to Ramsay, including small detours to pick up Bob ad Baldrine and drop him at Port Lewaigue.

Figure 1.2 shows Janet's altered route with places to pick up and drop Bob marked by the small round icons showing an arrow and a suitcae.

Janet's new route now includes small detours to pick up Bob at Baldrine (see figure for closeup) and dropping Bob at Port Lewaigue (see figure 1.4 for closeup)

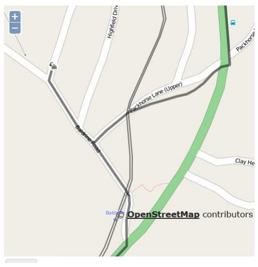

Figure 1.3: Janet's new route in detail, including the detour from main road A2 (green) to pick up Bob at Baldrine

1.7.2 Finding matching Offers for a given Request

Finding matching offers for a given request comprises the following steps:

1. For the request's starting point, determine all offers that have route-points coming within distance "detour" to the starting point and match the temporal bounds.

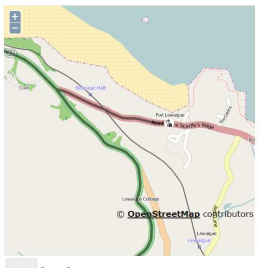

Figure 1.4: Janet's new route in detail, including the detour from main road A2 to drop Bob at Port Lewaigue

2. Determine all offers that come within range of "detour" to the drive's endpoint.

The intersection set of the requests in (1) and (2) is determined, and the offers in the intersection sets are filtered analogously to the processing at the final step of 1.7.1: First, the "cheap" filtering criteria (smoker, gender,..) are applied, and finally for the remaining request, the detour-route is calculated and it is checked if the detour is in the detour defined by the driver.

All offers that pass the final test are then presented to driver and rider for approval.

Chapter 2

Searching for Riders

In this section we give a more detailled description of the *"Search For Riders Algorithm"* which finds matching requests for a given offer. Searching for riders is done when a new offer gets created. this comprises the steps described in section 2.1 to 2.7:

2.1 Step 1: Create Route and Routepoints

First step when creating an offer is creating the *route* associated to the offer by calling the RouteMatchingBean.computeInitialRoutes(...) method.

Routepoints

The route is an ordered list of *routepoints*. Routepoints are modelled by the RoutePointEntity class, which has the following main properties:

Integer routeIdx Index of the routepoint inside the route (which is an ordered list of routepoints)

Point coordinate Spatial coordinates of the routepoint in longitude/latitude form

Timestamp expectedArrival Date/Time when the driver is expected to reach that point

Integer seatsAvailable Number of free seats the driver has availlable when reaching that point

Double distanceToSourceMeters Road distance to the startpoint in meters.

Henceforth, we may abbreviate the term *RoutePoint* with *RP*.

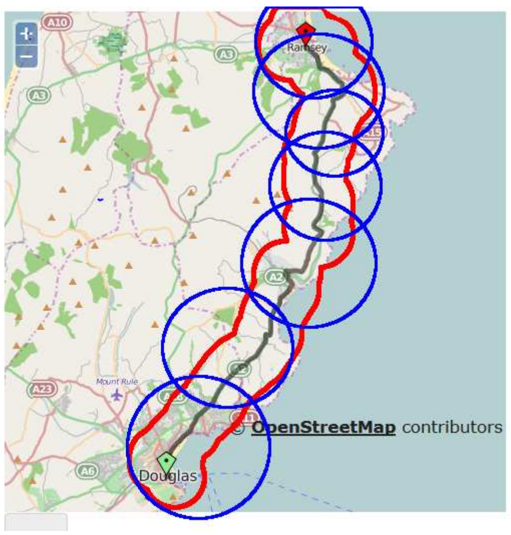

Figure 2.1: Red lines mark the exact corridor of distance $detour_{max}$ around Janet's route, The blue circles are a covering of the corridor as used by the circle overlay algoritm.

2.2 Step 2: Create Drive Route Points

As depicted in 1.7.1, in order to find a preselection of matching requests, a corridor consisting of all the points that have distance less than the driver defined maximum detour is scanned for start and endpoints of requests. The task of "scanning the corridor of distance max detour" is somewhat complicated and it's implementation is based on an auxiliary construction named "Drive Route Points", which we discuss next.

2.2.1 The Circle Overlay Algorithm

The principal Idea is to approximate the corridor of distance *detour* around the path by a number of so called *Drive Route Points* (henceforth abbreviated DRPs). The DRPs are chosen as a subset of the set of RPs. Instead of testing wether or not a point P is in the corridor, we test if there is a DRP which is close enough to P. Figure 2.1 depicts the circle overlay algorithm.

As the surface covered by these Circles is larger than the corridor, the circle overlay algorithm produces more false positives as checking the exact corridor would do. However, beeing used at the preselection stage, this step is supposed to produce only a preliminary list of results, which may contain a number of false positives which are removed at subsequent stages of the search for rider algorithm. The advantage in speed and effort of using circle overlay instead of the exact corridor anyway outweights this disadvantage.

2.2.2 Drive Route Points

In ORS, DRPs are modelled by the *DriveRoutePointEntity* class. These are the main properties of a DriveRoutePointEntity:

Integer routeIdx Index of the routepoint inside ordered list of DRPs)

Point coordinate Spatial coordinates in longitude/latitude form

Timestamp expectedArrival Date/Time when the driver is expected to reach that point

Integer seatsAvailable Number of free seats the driver has availlable when reaching that point

Double distanceToSourceMeters Road distance to the startpoint in meters along the route.

Double testradius Used in COA, see discussion of parameters below

2.2.3 Parameters for the Circle Overlay Algorithm

The following parameters are used in the implementation of the COA:

d The distance between each DRP

t_r The testradius. This is a radius around the DRP which guarantees that any point in the corridor which has direct distance $\hat{d}$ with $\hat{d} \leq \frac{d}{2}$ is inside of the circle of radius t_r around DRS. The testradius t_r is individual for each DRP.

Choosing a suitable distance between DRPs d and assigning an optimal testradius t_r are delicate tasks. While d should be chosen in a way that eliminates unneccessary calculations, given a choice of d, t_r has to be chosen such that all points in the corridor are covered.

Choosing t_r is discussed in much depth in section 4, choosing d is discussed in much depth in section 5.

2.3 Step 3: Scanning the Corridor

With the DRPs in place, the next step is to scan the set of requests for request *req* for which:

1. There exists a DRP drp_1 for this offer such that the startpoint of req is within the testradius of drp_1
2. There exists a DRP drp_2 for this offer such that the endpoint of req is within the testradius of drp_2
3. The expected arrival of drp_1 is within the limits of $req.startTimeEarlies$ and $startTimeLatest$

All requests conforming to the above criteria are put together into a preliminary list of matches (preselection), which is then subject to further filtering.

2.4 Step 4: Filtering by simple Criteria

The preselection is filtered for simple criteria (smoker/gender/...etc), matches that do not match all conditions are removed from the preselection. Note that this step does not require expensive calculation, so it should be performed before calculating and checking the exact detour.

2.5 Step 5: Calculating the Detour

For each request in the preselection list that passed the previous step, calculate the corrected tour, which includes the detour needed to pick up and drop the potential rider. Remove all those requests from preselection for which the detour is larger than the maximum detour defined by the driver.

2.6 Step 6: Applying the Sorting Function

Sort the remaining requests according to the scoring function.

2.7 Step 7: Applying Limits

From the sorted list of requests, return the top *ml* best scoring elements, Where *ml* is the drivers individual "matchLimit", i.e the maximum number of matching requests defined for the driver

Chapter 3

Searching for Drivers

In this chapter we cover finding matching offers for a given request. Searching for drivers is done when a requests gets newly created.

3.1 Step 1: Preselecting Offers

Search DRPs for Offers, which conform to the following criteria:

1. There exists a DRP for this offer,
   which has direct distance smaller than DRPs testradius to request's startpt
   AND
   DRP has seatsAvaillable more or equal to the request required number of seats
   AND
   DRP's expected arrival is within request's startTimeEarliest and startTimeLatest.

2. There exists a DRP for this offer,
   which has direct distance smaller than DRPs testradius to request's endpt
   AND
   DRP has seatsAvaillable more or equal to the request required number of seats

Offers matching the above tests are added to a *preselection list*. The follwing steps will be about subsequently removing false positives from the preselection list to finally obtain a set of exact matches in the end.

3.2 Step 2: Filtering by simple Criteria

The preselection is filtered for simple criteria (smoker/gender/...etc). Matches that do not match all of these conditions are removed from the preselection. Note that this step does not require expensive calculation, so it should be performed before calculating and checking the exact detour.

3.3 Step 3: Calculating the Detour

For each offer in the preselection list that passed the previous step, invoke the routing engine to calculate the corrected tour, which includes the detour needed to pick up and drop the potential rider. In this step all offers for which the detour is larger than the maximum detour defined by the driver are removed from the preselection list.

3.4 Step 4: Applying the Sorting Function

Sort the remaining offers according to the scoring function.

3.5 Step 5: Applying Limits

From the sorted list of requests, return the top *ml* best scoring elements, Where *ml* is the riders individual "matchLimit", i.e the maximum number of matchings defined to be returned for the rider.

Chapter 4

Choosing the Test Radius around a DRP

In this section we discuss the choice of the so called "testradius" which is assigned to every Drive_Route_Point. During this section, we assume that there is a fixed distance named d between the Drive_Route_Points in question. The distance d is not measured directly ("as the crow flies") but along the path between the DRPs.

We do not discuss the choice of d here, as this will be discussed in chapter 5.

4.1 Deflection from the direct Line

Given two DRPs d_1 and d_2 and a path between these points, by "Deflection" we denote the maximum distance between the Path and the direct line between d_1 and d_2. Figure 4.1 illustrates the deflection of a path. We will now discuss the "Maximum Deflection" from the direct line between to DRPs.

4.1.1 Maximum Deflection appears at Paths with *exactly one* route point

Keep in mind that a "Path" is composed of a finite number of RPs which are in turn connected by direct lines. Given DRPs d_1 and d_2 and all paths of given length d (and with arbitrary number of route points), we will now show that maximum deflection will appear at those paths which have *exactly one* route point.

More precisely, for every Path P leading from d_1 to d_2 with RPs $rp_1, rp_2, ..., rp_n$ with deflection h and length d, we can construct a path with $n-1$ routepoints

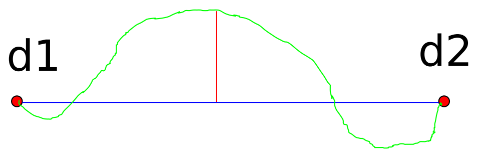

Figure 4.1: Deflection of a path. The deflection (red) is defined as the maximum distance of the path (green) from the direct line (blue)

and deflection $\tilde{h}$ >h as follows:

1. From each triple of route points rp_{j-1}, rp_j, rp_{j+1} where the three points are situated on one direct line, eliminate the middle point rp_j. Note that this will not change the geometry of the path. Iterate this process until there are no more three adjacent points on one line are left on the path.

2. From RPs $rp_1, rp_2, ..., rp_n$, chose an rp_j such that the maximum deflection h is realized at rp_j, and that there is one adjacent route point (wlog: $rp_{(j+1)}$) such that the distance from $rp_{(j+1)}$ to the direct line between d_1 and d_2 is smaller than h.

3. Delete rp_{j+1}. Since there are no three points located on a line, the resuling path $rp_1, rp_2, ..., rp_j, rp_{j+2}, .., rp_n$ is now shorter than the original path $rp_1, rp_2, ..., rp_n$

4. It is now possible to replace rp_j with a $\tilde{rp_j}$ further away from the direct line such that the resulting Path $\tilde{P}$: $rp_1, rp_2, ..., \tilde{rp_j}, rp_{j+2}, .., rp_n$ has length d, and deflection greater than that of the original Path P, as desired.

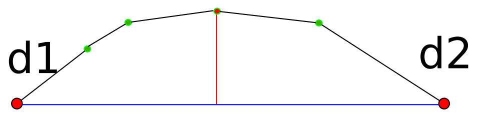

Figure 4.2: Depicting construction in 4.1.1: Original Path. Route point r_j realizing the maximum deflection and maximum deflection in red.

4.1.2 The Ellipse

As a consequence of section 4.1.1 we can now assume that a maximum deflection (maximal with respect to all paths of given length d) is realized by a

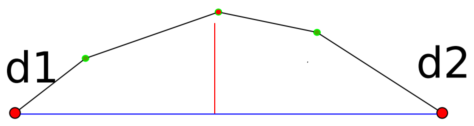

Figure 4.3: Depicting construction in 4.1.1: Route point r_{j+1} now deleted. Newly constructed RP $\tilde{rp}_j$ marked red. Original deflection marked by red line.

Path with only one single RP. Given d_1 and d_2, and a fixed d all the possible route points rp have the common property that the sum of the distances from rp to d_1 and d_2 is d. As this is exactly the defining property of the ellipse, all rp are located on an ellipse with focal points d_1 and d_2.

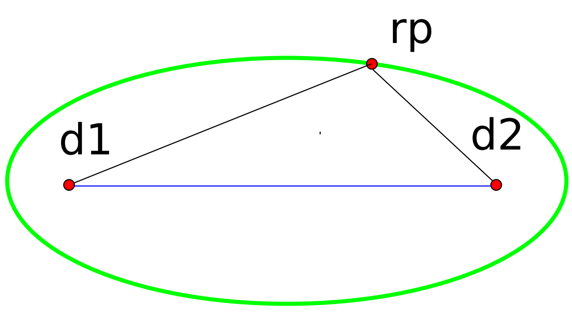

Figure 4.4: For all paths with just one single route point rp and with given length d, all possible locations or rp are located on an eclipse with focalpoints d_1 and d_2

4.1.3 The Maximum Deflection as Function of the direct Distance between Drive_Route_Points

In view of the fact that an eclipse is symmetric, it becomes clear that for fixed d, the maximum deflection becomes apparend exactly in the middle between d_1 and d_2. With $\tilde{d}$ defined as the direct distance between d_1 and d_2, applying the Pythagorean Theorem, the maximum deflection h_{max} can be easily calculated to be:

$$h_{max}(\tilde{d}, d) = \sqrt{\left(\frac{d}{2}\right)^2 - \left(\frac{\tilde{d}}{2}\right)^2} \tag{4.1}$$

Equation 4.1 gives also an upper bound h_{max} for $h_{max}(\tilde{d})$, which is independent of $\tilde{d}$, and may be easier to calculate in some situations:

$$h_{max} = \frac{d}{2} \tag{4.2}$$

Note also, that the independent bound in equation 4.2 is not completely unrealistic. For example if points d_1 and d_2 are divided by some obstacle such as a railway or a river, then $\tilde{d}$ may come close to zero, and the bound in given in 4.2 gets realized.

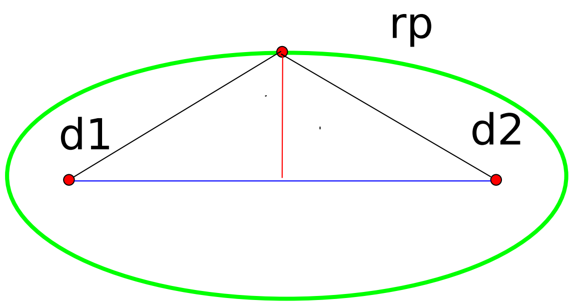

Figure 4.5: Depicting use of the Pythagorean Theorem to derive the formula for h_{max}: $\frac{\tilde{d}}{2}$ (blue), h_{max} (red) and $\frac{d}{2}$ form (two) rectangualar triangles

4.2 The Testradius

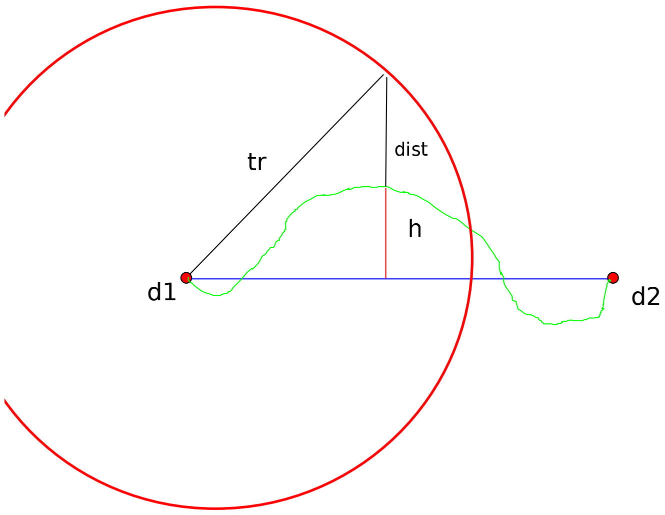

Figure 4.6: Testradius t_r must be large enough so that it covers any deflection h *plus* the user defined distance *dist*.

Summing up the results of the preceeding section, we now present formulas for calculating the test radius t_r.

We now consider two DRs d_1 and d_2 connected by a path P, with the distance from d_1 to d_3 beeing d along P and $\tilde{d}$ in direct line. Assume further that P has a deflection h. By convexity we can assume this deflection to appear at the middle of the line segment between d_1 and d_2. I.e: if the radius is large enough to cover the deflection in the middle of the line segment, this deflection will be covered anywhere else.

Now the testradius tr around d_1 must be large enough to cover a distance of $h + dist$ at a distance of $\frac{\tilde{d}}{2}$ from d_1. The Phythagorean Theorem then gives the condition:

$$t_r^2 \geq \left(\frac{\tilde{d}}{2}\right)^2 + (h + dist)^2 \tag{4.3}$$

Condition 4.3 can be fulfilled with equality by choosing a radius t_r as the square root of the right side:

$$t_r := \sqrt{\left(\frac{\tilde{d}}{2}\right)^2 + (h + dist)^2} \tag{4.4}$$

In practical application, it may turn out to be too tedious to calculate the exact deflection h of the path P, so we can estimate the exact value h by the upper bound $h_{max}(\tilde{d}, d)$ from equation 4.1. This yields a larger valid testradius t_{r1}, which also fulfills 4.4, and has a more simple formula:

$$t_{r1} := \sqrt{\left(\frac{\tilde{d}}{2}\right)^2 + \left(\sqrt{\left(\frac{d}{2}\right)^2 - \left(\frac{\tilde{d}}{2}\right)^2} + dist\right)^2} \tag{4.5}$$

Not subtracting the term $\left(\frac{\tilde{d}}{2}\right)^2$ in the inner square root gives another valid choice tr_2 for a testradius:

$$t_{r2} := \sqrt{\left(\frac{\tilde{d}}{2}\right)^2 + \left(\frac{d}{2} + dist\right)^2} \tag{4.6}$$

Finally, as $\tilde{d}$ is bounded by d, a really simple formula for a valid testradius t_{r3} can be obtained, from 4.6, which does not even require calculation of the direct distance $\tilde{d}$:

$$t_{r3} := \sqrt{\frac{1}{2}d^2 + d * dist + dist^2} \tag{4.7}$$

Chapter 5

Choosing the Distance between DRPs

In this section we present some considerations on how to choose the distance d along the path between two DRs.

5.0.1 The Defect

A general measure and formula for the defect can be derived from figure 5.1: All positives are located inside the circle of testradius t_r around d_1. Of these, the *"true positives"* are contained in the rectangle F with height $2 * h + 2 * dist$ and width $\tilde{d}$, where $\tilde{d}$ is the maximum of the direct distance between d_1 and either its left or right neighbor DR. To get a measure for the expected defect, we have to calculate the difference between the measure of the circle containing *all positives* and the rectangle containing the *true positives*:

$$D := \pi * {t_r}^2 - F \tag{5.1}$$

From figure 5.1 the identity $F = (\tilde{d} * 2(h + dist))$ can be read. When this is substituted into the definition 5.1, the definition takes on the form:

$$D := \pi * {t_r}^2 - (\tilde{d} * 2(h + dist)) \tag{5.2}$$

In the rest of this section, we discuss how the ratio between true positives and false positives can be minimized by a sensible choice of the parameter d.

Assuming that interesting points are uniformly distributed in the plane, the ratio between D and F is a measure for the ratio between all positives and false positives. This gives:

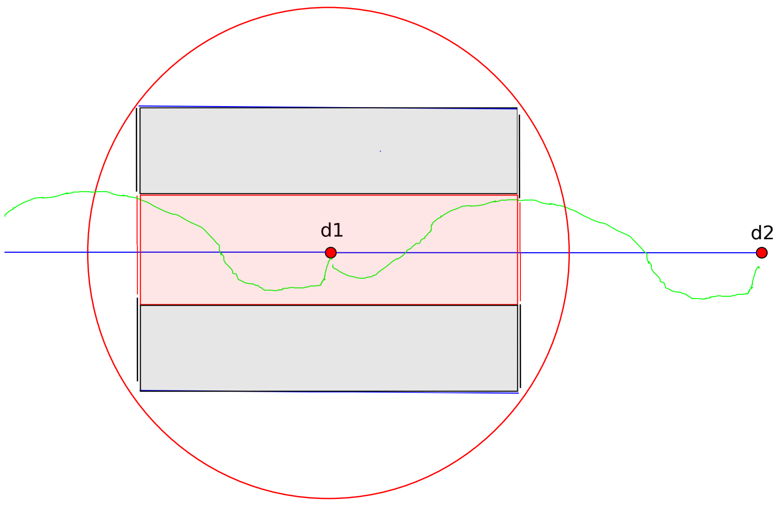

Figure 5.1: Demonstating the *"Defect"*: All points within the *red bar* have a distance not greater than h from the direct line, while all the points inside the gray bars have distance greater than h, but less than $h + dist$. All points locate inside either grey or red bars can be considered a success, while points located inside the circle of radius t_r but outside the bars can be considered "false positives".

$$\frac{D}{F} = \frac{\pi * {t_r}^2 - (\tilde{d} * 2(h + dist))}{(\tilde{d} * 2(h + dist))} \tag{5.3}$$

Hoever simple formula 5.1 may seem, keep in mind that the formal parameters t_r and $\tilde{d}$ may themselves depend on the choice of d, while the deflection h may vary between 0 and d and depends on geography – which is largely out of *our* control too.

For this reason, in what follows we discuss the optimal choice of d for some special cases only.

5.1 Choosing d for a straight Path

In this section, we discuss the choice of d in case of a straigh path. For a straight path, we have that the deflection is zero, i.e:

$$h = 0 \tag{5.4}$$

and that the direct line between two DRPs is equal to the path between these DRPs, i.e:

$$\tilde{d} = d \tag{5.5}$$

Such that the formula for the general defect D simplifies to

$$D := \pi * {t_r}^2 - (d * 2 * dist) \tag{5.6}$$

While F becomes

$$F := (d * 2 * dist) \tag{5.7}$$

Giving the following simplified formula for $\frac{D}{F}$:

$$labelDF_straight_line\frac{D}{F} = \frac{\pi * {t_r}^2 - (d * 2 * dist)}{d * 2 * dist} \tag{5.8}$$

Where the formal parameter t_r might still implicitely contain d !

5.1.1 Optimal Choice of d for Straight Line and Testradius t_{r3}

Substituting the t_{r3} as defined in , formula **??** becomes:

$$\frac{D}{F} = \frac{\pi(\frac{1}{2}d^2 + d * dist + dist^2) - (d * 2 * dist)}{d * 2 * dist} \tag{5.9}$$

Which becomes minimal for choosing d as:

$$d := (1 - dist) + \sqrt{1 - 2 * dist + dist^2 - \frac{d * dist^2}{\pi}} \tag{5.10}$$

Bibliography

[OpenRideRoutenMatching] *Open Ride Routen Matching*,
(no author given, in German)
Fraunhofer Institut für offene Kommunikationssysteme
(no author given, in German)
online at http://sourceforge.net/p/openride/code/HEAD/tree/trunk/doc/Architecture/I
RoutenMatching-doc-02-03-11.pdf?format=raw

Index

Approval, 9

Circle Overlay Algorithm, 11
    Parameters, 12
Corridor, 7
    Scanning for Requests, 13
    Search for Riders, 11

Detour, 8, 9
    Search for Drivers, 15
    Search for Riders, 13
Drive Route Point, 12
    Distance between DRPs, 12
    Properties of, 12
DRP, 12
Dynamic Rideshareing, 4

False Positives
    Search for Drivers, 14
Filtering Criteria (cheap), 9

Match Limit
    Search for Riders, 13
Match Limits
    Search for Drivers, 15

Offer, 5
    finding matching offers for given request, 8, 14
    finding matching requests for given offer, 7, 10

Preselection List
    Search for Drivers, 14
    Search for Riders, 13

Request, 6
    finding matching offers for given request, 8, 14
    finding matching requests for given offer, 7, 10
Rideshareing, 4
Route, 6
Route Matching, 5, 7
Routepoint, 6, 10
RP, 10

Scoring Function
    Search for Drivers, 15
Simple Criteria
    Search for Drivers, 15
    Search for Riders, 13
Sorting Function
    Search for Riders, 13

Testradius, 12