Route Matching Algorithm

Gleb Chuvpilo[1]*, Sanny Liao[2], Oscar Salazar[3], Sergio Botero[4]

Abstract
This technical report describes the *Schedule Matching Algorithm* and *Route Matching Algorithm* that we propose to form rides in *Ride Marketplace v1*. It will replace the current *Simple Agglomerative Clustering Algorithm* used in *Ride Marketplace Alpha*. The difference between the two is that *Ride Marketplace v1* takes into consideration varying amounts of overlap between a potential driver and a potential passenger's desired travel schedule; and instead of using great circle distance between two latitude/longitude points to group users, the new algorithm produces optimal routes that originate at a driver's home location, visit all of the driver's passengers in the most efficient order, and end at the work location, all while providing global optimality. Finally, we propose metrics for measuring algorithm quality. They define objective criteria to allow apples-to-apples comparisons across algorithms, and enable future research in the space.

Keywords
Matching — Clustering — Scheduling

[1] *Vice President of Research and Development, Ride*
[2] *Data Scientist, Ride*
[3] *Chief Product and Technology Officer, Ride*
[4] *Software Engineer, Ride*
***Corresponding author**: gleb@ride.com

Contents

Introduction

This technical report describes the *Route Matching Algorithm* and *Schedule Matching Algorithm* and that we propose to form rides in *Ride Marketplace v1*. It will replace the current *Simple Agglomerative Clustering Algorithm* used in *Ride Marketplace Alpha*. The new algorithm builds on the *Simple Agglomerative Clustering Algorithm* to produces optimal routes that originate at a driver's home location, visit all of the driver's passengers in the most efficient order, and end at the work location. Furthermore, the new algorithm provides a mean to handle repeated trips by taking into account the amount of overlap between two traveler's desired schdule. Finally, we propose metrics for measuring algorithm quality. They define objective criteria to allow apples-to-apples comparisons across algorithms, and enable future research in the space.

1. Route Matching Algorithm

1.1 Overview

The pseudocode for Route Matching Algorithm is shown in algorithm 1.

1.2 Detailed Description

We start our route matching by pre-computing directions from each of the users to target destination (line 2). We then order the list of drivers by decreasing distance to target destination (line 3). This is the greedy aspect of the algorithm: we guarantee that we start with the furthest driver, and then move closer with every step. After this step we store vehicle capacities (line 4). We then store all passengers into another set (line 5). Finally, to conclude algorithm setup, we reset the list of

Algorithm 1: Route Matching

input : U Users and T Target destination
output : R Routes

1 **begin**
2 $M \longleftarrow Distance(U_i, T)$
3 $D \longleftarrow FindDrivers(U).sortDecreasingBy(M)$
4 $C \longleftarrow FindCapacity(D)$
5 $P \longleftarrow FindPassengers(U)$
6 $R \longleftarrow \emptyset$
7 **for** $d_i \in Ds.t.M(d_i) > M(d_j), \forall j$ **do**
8 **while** $TotalPassengers(d_i) < c_i$ **do**
9 $W \longleftarrow DirectionsViaWaypoint(d_i, r, U_i, T)$
10 $r \longleftarrow P.sortIncreasingBy(W).first()$
11 $return(R)$

routes (line 6).

The core of the Route Matching algorithm is a loop that selects the furthest remaining driver (line 7). For this driver, we start building up an array of passengers in the following fashion: first, we calculate additional travel time if we add each one of the passengers (one at a time), and choose the passenger that adds the least additional travel time. We then continue attempting to add additional passengers (again, one a time) while each time computing an optimal pickup route by solving the so-called *Traveling Salesman Problem*. We continue the procedure until we either run out of capacity, or run out of passengers that satisfy our business rules (for instance, the total additional pickup time should not exceed x minutes). We then save this route, and remove both the driver and the passengers from the list of available (lines 7-10).

1.3 Optimization

Due to the fact that solving the *Traveling Salesman Problem* is a very intensive computational procedure, we can prune the search space by running an agglomerative clustering filter [1] before attempting to add passengers to a driver (lines 7-10). This will significantly speed up the running time of the algorithm.

1.4 Asymptotic Complexity Analysis

Without optimization, the algorithm solves the *Traveling Salesman Problem* $O(n^2)$ times, where n is the number of users. This is because the algorithm has to process $O(n)$ drivers, and for each one attempt to add $O(n)passengers$ up to c times, where c is capacity. With the optimization, we limit the search space to a constant, which reduced complexity to calling the *Traveling Salesman Problem* solver $O(n)$ times.

1.5 Implementation Details

We use third-party geocoding [2] and directions [3] services, as well as a solver of the *Traveling Salesman Problem* [4].

2. Schedule Matching Algorithm

2.1 Overview

Schedule mismatch poses an equally important challenge to carpooling. When users have non-recurring trips, the algorithm accounts for schedule mismatch using a mask to find users with compatible anchor times for the trips concerned, where anchor time can either be arrival time or departure time. When users have recurring trips in a given time span, in addition to schedule mismatch for a single trip, the algorithm also takes into consideration the frequency at which two travel schedules are compatible. Optimizing for schedule mismatch necessities a compromise in route optimality, we describe a simple method to parametrize the tradeoffs.

2.2 Anchor time

Depending on the direction of travel, each traveler has an anchor time in the form of either desired arrival time or departure time. Suppose a driver has an anchor time of A_j, we fix the driver's anchor time and look for a set of passengers whose anchor time falls within x minutes of the driver's anchor time. In other words, driver j has a mask M_j:

$$M_j = [A_j - x, A_j + x] \quad (1)$$

For each driver j, we evaluate each user who can be a passenger against the mask M_j to createa a consideration set of passengers C_j. For non-recurring trips, C_j can be used directly by the *Route Matching Algorithm* to construct optimal carpools around each driver.

2.3 Overlap score

For users with recurring trip, we construct an overlap score to capture the degree to which a passenger's desired set of trip overlaps with that of a potential driver.

Let T_i be passenger i's desired schedule, and T_j be driver j's desired schedule, where

$$T_i = [s_i(t_1), s_i(t2), ..., s_i(t_S)]$$
for a passenger with S desired trips (2)

$$T_j = [s_j(t_1), s_j(t_2), ..., s_j(t_T)]$$
for a driver with T desired trips (3)

The overlap of passenger i's schedule to that of driver j's can be measured as:

$$OverlapScore_j(passengeri) = \chi(T_i, T_j) \quad (4)$$

2.4 Distance score

To proxy for route optimality, we construct a distance score for each potential passenger relative to a driver. Given destination address D_d, passenger address D_i, and driver address D_j, the distance score for each potential passenger i with respect to driver j is:

$$DistanceScore_j(passenger_i) = \omega(D_d, D_i, D_j) \quad (5)$$

2.5 Eligibility score

We parametrize the tradeoff between schedule overlap and route optimality using a function with overlap-score and distance-score as inputs.

$$EligibilityScore_j(passenger_i) = \psi(OverlapScore_j(i), DistanceScore_j(i)) \quad (6)$$

Eligibility score is increasing in both overlap score and distance score. The relative effect of either overlap score or distance score on eligibility score may be fixed, or may vary based on outside factors, such as travel times, relationship between two carpoolers, or other individualized characteristics. Using the eligibility score, we rank all users in the consideration set C_j for each driver, and use the top n poptential passengers as inputs for the *Route Matching Algorithm*.

3. Measuring Algorithm Quality

3.1 Overview

We would like to be able to quantify improvements in each algorithm update going forward. There are three main ways in which we would like to measure algorithm quality:

1. Post-match trips in a day
2. Probability of match acceptance
3. Expected trips in a day

We propose to start using metric (1) and a simplified version of metric (2) right away. In the meantime, we should start collecting data on other components for (2) and (3) for future improvements.

3.2 Post-match trips in a day

The macro goal here is to reduce the number of cars on the road. We want to measure that. Let's define a trip as one driver who drives from point A to point B in a day, irrespective of whether a driver has passengers or not. The following example provides three scenarios[1] :

In this example:

- Let T_0 be the number of trips that would be taken for a fixed number of employees to commute from home to work and back when traveling alone

[1] *Travel Alone* is a base case when none of the employees carpool

Table 1. Scenario Analysis

	Travel Alone	Algo 1	Algo 2
Number of Trips	T_0	T_1	T_2

- When Ride pools employees into carpools, T_s trips are formed, where $T_s \leq T_0 \forall s \neq 0$
- Algorithm 2 is superior to Algorithm 1 iff $T_2 < T_1$

3.3 Probability of users accepting match

We want to maximize the probability that users will accept the matches that we have formed for them. To do this, we need to consider a few factors:

- Utility of saving money to the user, $U(dollar)$
- Utility of driving or riding with a fellow carpooler, this can include everything from companionship to environment impact, $U(carpool)$
- Utility of driving or riding a different car over his/her own, $U(cartype)$
- Utility of the inconvenience that carpooling produces, $U(inconvenience)$

Putting them together, we have:

$$Prob(\text{user i accepts match}) = Prob(U_i(dollar) + U_i(carpool) + U_i(cartype) > U_i(inconvenience)) \quad (7)$$

For the time being, we will stay agnostic toward how each of our users value dollar, carpool, car type, and inconvenience. However, we will try to measure inconvenience as a function of difference in schedule and travel time. Generically, inconvenience can be modeled as:

$$Inconvenience = \alpha * F(\Delta traveltime) + \beta * G(\Delta schedule), \quad (8)$$

where $\alpha + \beta = 1$
Functions F and G are defined as:

$$F(\Delta traveltime) = \ln(\tfrac{TTWC - TTWDA}{TTWDA} + 1) \quad (9)$$

where

- $TTWC$ is *travel time when carpooled*
- $TTWDA$ is *travel time when driving alone*

$$G(\Delta schedule) = \ln(\tfrac{\text{DAT-CAT}}{(60 \text{ minutes})} + 1) \quad (10)$$

where

- *DAT* is *desired anchor time*
- *CAT* is *carpool anchor time*

In summary, instead of comprehensively modeling the probability that a user will accept a match, we will use a metric that simply captures the inconvenience that a match would have causes a user. We will operate under the assumption that the less inconvenient a match is, the more like he/she will accept the match.

3.4 Expected trip in a day

Given T_s, the number of trips that a matching algorithm forms in a day, and the probability that each user accepts the match formed for her, we can comprehensively capture the quality of an algorithm as the expected number of trips to take place each day:

$$\begin{aligned}&\text{Quality of algorithm m} =\\ &\text{Expected number of trips in a day} =\\ &\qquad \sum_{1}^{T_m} 1 \times Prob(\text{users in ride i accept match}) \quad (11)\end{aligned}$$

where T_m = the total number of trips formed by algorithm m

References

[1] Heather Arthur. K-means and hierarchical clustering. `https://github.com/harthur/clusterfck`, 2012. [Online; accessed 24-September-2014].

[2] Google Inc. Geocoding API. `https://developers.google.com/maps/documentation/geocoding/`, 2014. [Online; accessed 24-September-2014].

[3] Google Inc. Directions API. `https://developers.google.com/maps/documentation/directions/`, 2014. [Online; accessed 24-September-2014].

[4] Google Inc. Traveling Salesman Solver. `https://developers.google.com/maps/documentation/directions/#Waypoints`, 2014. [Online; accessed 24-September-2014].