Reduce your consumption, ecological footprint, costs, and energy dependency.

Share and cooperate. Cooperation has a way of benefiting everyone involved. Ride with your friends, befriend your riders.

Hundreds of millions of trips are taken every day from places we all live to places we all go.

The energy costs and global consequences of overconsumption don’t seem to be falling.

Add your trip to the lot, and find someone going your way. Enter your ride locations, schedule, and preferences. If we find matching requests, we will inform you by email and text. It’s that simple.

In addition to providing you with results, we generate matches in a way that allows more people to be matched. This means that we improve the likelihood that you get results, and reduce everyone’s ecological footprint even further. :)

Why This Problem is Difficult

The problem of finding the best possible overall driver and rider matching (i.e. global best solution) is just a version of a well-known problem called the Vehicle Routing Problem (VRP), which is stated as:

"The Vehicle Routing Problem or VRP is a combinatorial optimization and nonlinear programming problem arising to service a number of customers with a fleet of vehicles. Proposed by Dantzig and Ramser in 1959, VRP is an important problem in the fields of transportation, distribution and logistics. Often the context is that of delivering goods located at a central depot to customers who have placed orders for such goods. Implicit is the goal of minimizing the cost of distributing the goods. Many methods have been developed for searching for good solutions to the problem, but for all but the smallest problems, finding global minimum for the cost function is computationally complex."

—Wikipedia entry for Vehicle Routing Problem

It is classified as an NP-Hard problem, meaning that no known polynomial-time algorithms for solving it exist. If we approached solving it in the naive sense, i.e. try every possible solution, reaching a solution would take an amount of time that exceeds the history of the known universe.

To top it off, Rideshare includes scheduling and preference matching, and allows for arbitrarily many depots.

Our approach involved using three different algorithms which were developed by many of our wise predecessors, and adapted by us.

How We Find an Optimal Solution

In order to match users, we utilize three different Optimizers. An Optimizer takes in a set of drivers and riders, and creates a “seed solution.”

A Solution sets up Rideshares which initially contain only the drivers.

The Optimizer applies an algorithm to find an optimal result. It compares different solutions using a scoring function, and generates the highest scoring solution it can, returning it as the optimized solution.

The score of a given solution is:

\[ \text{Total Score} = \frac{\text{Number of Riders Matched} + \text{Number of Drivers Matched}}{2 \times \text{Drivers' seed distances} + 2 \times \text{Total final distance}} \]

The whole part of the score represents the primary criteria for comparing solutions. It consists of the total number of users matched.

Our secondary criteria for a good solution is the fraction. It gives us an idea of the average driver deviation from their original route. A smaller fraction means shorter deviations.

Compatibility - Can I Add This Rider?

How to take a Rideshare and rider and ascertain whether they are compatible. This might sound like a simple enough task, but it involves much more than meets the eye.

Of course, checking for age and gender preferences is simple enough. But schedule and route compatibility are a different story. It turns out that the first can be done if we apply a clever observation about the Driver’s worst case leave window, and the second is yet another instance of our good friend the Vehicle Routing Problem.

Suppose we have Marisa Driver and Joe Rider in a match, and we would like to see if adding a Alex Rider would create an incompatible ride schedule or route length.

To do this we need to find out the best way of ordering the riders, so as to minimize route length. Since the capacity of a vehicle is fairly small, we find the best route by simplifying all combinations.

Once we have an optimal route for this group, we verify that the driver any doesn’t conflict with anyone’s schedule. We do this by narrowing the leave and arrive windows until we reach a worst case leave window for the driver.

We then add to this leave window the time necessary to get from point to point, each time verifying that the users’ time windows do not conflict with the arrival window for that point.
Pseudo-code For the Algorithms

**Bipartite Matching Optimizer**

1. Get seed solution
2. Construct compatible rideshares and ride requests with unmatched edges.
3. For a random unmatched rider, check if there is a path that connects it to an unmatched edge.
4. Repeat 0-3 until all unmatched riders have been tried.

**Genetic Optimizer**

1. Get seed solution
2. Spawn a generation of solutions from the current population.
4. Keep a copy of the best solution.
5. Spawn another generation from the last generation's best solutions.
6. Repeat steps 2-5 for each generation.
7. Once the score settles, return the best solution.

**Brute Force Optimizer**

1. Get seed solution
2. For each solution in the space of all possible solutions, compare it with the best one so far.
3. Keep only the better of the two.
4. Return the best solution.

---

**Scores**

**Genetic vs. Bipartite**

- Genetic
- Bipartite

---

**Number of Requests vs. Run Time**

- Bipartite
- Genetic

---

**Number of Requests vs. Natural Logarithm of Run Time**

- Bipartite
- Genetic

---

**Run Time vs. Final Score**

- Bipartite
- Genetic
Implementation

Optimizer
- optimize()
- getBestScore()
- getBestSolution()
- iterate()

BruteForceOptimizer
- optimize()
- getBestScore()
- getBestSolution()
- iterate()

GeneticOptimizer
- optimize()
- getBestScore()
- getBestSolution()
- generatePopulation()

BipartiteOptimizer
- optimize()
- getBestScore()
- getBestSolution()
- single_optimize()
- iterate()
- getAugmentingPath()
- updateSolution()
- updateEdges()
- initEdges()
- riderIterate()
- matchIterate()

Optimizers:
- Optimizer
- BruteForceOptimizer
- GeneticOptimizer
- BipartiteOptimizer

Optimizer Structs:
- CompatibilityMatrix
- RouteMatrix
- RouteMatrixLocal

Testing:
- TesterUtil
- TestCases

Utility:
- SolutionUtils
- Util
- TesterUtils

Class List
- Address
- BipartiteOptimizer
- BruteForceOptimizer
- CompatibilityMatrix
- DBAccess
- DriverRouteSubmission
- GeneticOptimizer
- Location
- Match
- Optimizer
- PointInfo
- Preference
- RiderRouteSubmission
- Rideshare
- RouteClient
- Route
- RouteInfo
- RouteMatrix
- RouteMatrixLocal
- RouteOptimizer
- RoutePreference
- RouteSubmission
- Solution
- SolutionUtils
- TestCases
- TesterUtil
- TimeWindow
- UserInfo
- UserPreferences
- Utility

Route
- startLoc()
- endLoc()
- getLoc()
- numLocs()
- get_type() = 0
- UserPreference userPref
- RoutePreference routePref

DB Structs:
- TimeWindow
- UserInfo
- Solution
- Match
- Address
- Location
- PointInfo
- Preference
- Rideshare

RouteSubmission
- setComment()
- setUserInfo()
- get_type()
- uint32_t routeId
- uint32_t userId
- double optimalDistance
- UserInfo userInfo

RiderRouteSubmission

DriverRouteSubmission
- size_t capacity