A novel journey sharing platform for collective events

Ricardo Jesus  
Instituto Superior Técnico  
Lisboa, Portugal  
ricardo.n.jesus@tecnico.ulisboa.pt

João Barreto  
Instituto Superior Técnico  
Lisboa, Portugal  
joao.barreto@tecnico.ulisboa.pt

ABSTRACT
This dissertation explores the implementation of a platform that focuses on collective events, which are distributed amongst acquaintances, and calculates the best clusters of participants. This focus renders mainstream solutions inappropriate by changing the paradigm to n-to-n matchmaking. Several different methodologies and technologies were analyzed in order to provide an efficient and scalable solution capable of providing the best route possible (or close to best). Users will interact with the platform via a mobile application where they input constraints for the event. Subsequently, the back-end will cluster users together using a combination of heuristics and combinatorial optimization solution to minimize the distance traveled and the number of cars. With our evaluation we found that combinatorial optimization solutions provide the best results. However, for events of larger size, heuristic approaches excel with faster execution times.

KEYWORDS
Ride-sharing; Collective events; Mobile application; Clustering; Heuristics; Combinatorial optimization;

1 INTRODUCTION
Motivation
In an ever increasing automotive world we can not help but watch as it exacerbates long existing problems and concerns. As of publishing this document, the transportation sector is a leading cause of greenhouse gas emissions[1–3]. To many, commuting by car comes not as a privilege but as a necessity[4]. That necessity however, comes at a very high financial[5, 6] and psychological cost (long commutes, traffic congestions, overbooked parking)[7].

Ride-sharing becomes a promising, viable and distinct solution to reduce these concerns.

Ride-sharing consists of matching a driver with one (or several) riders in a (mostly) shared trip. Platforms can take into consideration constraints like pick-up and drop-off locations, time of departure and arrival, and shared route, taking them all into consideration when matching users. The concept is mostly applied to taxi services, transport of elderly and patient transportation.

Popular platforms like UberPool, Lyft Line or BlaBlaCar[8–10], follow a strict and rigid interaction between users. Users designate themselves as drivers or riders and they are exclusive. Most of the times, the drivers are company employees and it is the riders that are the product. This approach is compelling and attractive for sporadic travels of leisure or work and the occasional commute. But, it is a personalized and individual interaction and as such, it is inadequate for collective events and shared plans.

Objectives
Existing ride-sharing platforms employ a 1-to-n match approach, 1 rider (the application user) to n drivers (close-by registered drivers).

For this thesis we developed a platform that, given the focus on collective events, needs to calculate matches with an n-to-n approach (in contrast of 1-to-n). We face the problem of finding the best match for every participant. Riders need to be matched with drivers, furthermore, potential drivers must analysed to determine if they will actually drive to the destination, or go as a rider in someone else’s car. All users must be analysed and grouped, as no one must be left without transportation.

This thesis main objective is: To cluster users while optimizing common ride-sharing necessities. We seek a solution that minimizes the total distance travelled by the group, as well as the number of cars necessary to take the group to its destination.

To verify the performance of our clustering algorithm, we designed a mobile application for friends to plan their trips together. For a smooth and engaging interaction the clustering solution needs to provide efficient and scalable matching, returning the best route possible.

2 RELATED WORK
Common ride-sharing problems solve impromptu requests[11–14] from riders that wish to be matched with drivers. It is an individual, personal use of the service to share rides with other unknown users, a 1 rider to n existing drivers clustering solution. Given those platforms necessity to generate revenue, some solutions focus on monetary return[15–17].

Our platform aims to solve a new paradigm, with a different focus, where users ride-share with their acquaintances. There are several possible drivers (that volunteered when participating), but everyone can be a rider, meaning there are n riders and n driver, n-to-n clustering. To group participants
we studied and compared several methodologies, employing adaptations of existing work.

Heuristics
Existing academic studies developed several methodologies to match users and cluster riders with existing drivers. We studied several ride-sharing solutions. We developed three heuristics that implement common matchmaking implementations adapted from those solutions.

Route sharing: We calculate each user’s route to the destination. Then, users are matched based on how much route they share. This was devised from Mukherjee et al.’s [18] work that defined a solution to remove uncertainties and delays for riders by comparing riders destinations with their list of drivers’ routes.

Radius: We define a distance radius centered on each user and try to match them with other users that fall inside their radius. This heuristic was based on two solutions: 1) WEtransport, a modular, context-aware ride-sharing platform developed by Shirazi et al. [19] that allows for creating arbitrary clients with specialized UI to capture different contexts and constraints. Matches are found by defining a distance radius with time offsets. 2) The Scalable Dynamic Ride Sharing System (SHAREK) by Alarabi et al. [20] that was envisioned to extend existing ride-sharing systems. To match riders with drivers, it runs a range query on a grid to retrieve the drivers within a circular range centered at the rider location.

Voronoi cells: We match users based on their distance. Users are matched with the closest available user. This heuristic was used by Khan et al.’s [21] ride-sharing solution that tries to encourage ride-sharing by using intermediary meeting points, where one user can park its car and ride-share with the other. Voronoi cells are between users and meeting points, matching convergings users.

Combinatorial Optimization
In an attempt to group users according to common concerns and intentions, we studied approaches focused on minimizing two parameters: the number of cars and/or the distance travelled. This optimization can be achieved has combinatorial optimization problems.

Minimizing distances. To minimize the distance travelled we can adapt our problem into different existing solutions, that imply testing many possibilities in the search of the optimal one.

The Integer Linear Programming (ILP) defines the problem as a mathematical system and brute-forces the resolution of its mathematical system searching for the solution. It computes the best solution to a problem in which the objective function and the constraints (other than the integer constraints) are linear. An ILP formulation verifies every combination of constraints trying to achieve its objective function.

The Vehicle Routing Problem (VRP) describes the problem in a weighted, directed graph employing some heuristics to traverse the graph in search of the solution. To apply a VRP solution to Go-Together, a couple variations are necessary. Namely, a capacitated variation, because cars have a limited number of seats, and an open variation, because drivers start their route in their starting location and end it at the event’s destination. Also, drivers must be considered as possible pick-ups for other drivers.

Minimizing the number of cars. When trying to minimize the number of cars in an event, the problem becomes a packing problem, a class of optimization problems in mathematics that involve attempting to pack objects together into containers. The goal is to pack all objects using as few containers as possible.

In a bin packing problem, we are given containers and objects. Containers are a single two- or three-dimensional convex region. Objects, some or all of which, must be packed into one or more containers. The set may contain different objects with their sizes specified, or a single object of a fixed dimension that can be used repeatedly.

Many heuristics have been developed to solve the optimization problem. An analysis of some of these heuristics was independently conducted by researcher Bastian Rieck [22].

3 ARCHITECTURE
In this chapter we present the architectural representation of our work. Then we will describe each of the components of the model and its expected behaviour. We will explain the functionalities of the frontend and the backend, detailing the how it achieves its clustering capabilities.

Figure 1: Go-Together’s architecture.
Frontend
In the frontend, a user will be able to create and plan events and share them with the travelling group. Each participant of the event will fill in constraints and submit them. When desired, the event creator can request Go-Together to perform matchmaking with the enrolled users and assign riders to drivers. This request is sent to the backend, which responds with a planned route that is then displayed in an interactive map.

(1) Invitations: Upon event creation, the creator will have the option to share it. For sharing, the indicative, event-associated token will be showed. This token can then be passed on to other users (via external means of communication). The token receiver will then be able to introduce said token to join. The platform validates the token and allows the participant to visualize the newly joined event.

(2) Planned Route: Once the clustering and matchmaking is done (by the backend), the users will be able to see their assigned route on the frontend.

An interactive map will be shown to every cluster with a personalized route. This route will show the selected driver and riders, pick-up order and finalizing on the predetermined destination.

The frontend controls the access level of users in an event. While everyone can create and join events, only event creators can delete events and conclude them. There is no implemented invitation system, so if users save the event’s unique identifier they can share it with anyone, however, only event creators can access the identifier.

Backend
The backend is the core of the system. It is responsible for:
1) authenticating users, 2) storing, in a database, user information and event’s constraints, 3) perform the computations required to cluster users of an event and 4) calculate routes to the corresponding users.

(1) Authentication: Users need to be validated and authenticated to use the platform (section 4). Credentials are verified by our backend before synchronizing user’s data.

(2) Storage: The backend will manage the storage of users and events (section 3). All information is stored in a cloud database (as opposed to local storage) to make it distributed and accessible from different devices.

(3) Clustering: It is vital to match riders with drivers successfully. The necessary matchmaking is achieved with the use of a clustering algorithm. As explored in Sections 2 and 2, there are several possible approaches and algorithms. In chapter 5 we evaluate the different methodologies and in chapter 6 we wrap the implemented clustering mechanism.

Our backend simultaneously employs two approaches to cluster, one to minimize the total distance travelled by participants of the event and another to reduce the number of cars travelling to the destination. To achieve this, the clustering request sends two parameters: How much priority to be given to minimizing the number of cars ($cars_{param}$) and to minimize traveling distance ($distance_{param}$), where $cars_{param} + distance_{param} = 100$. Then, through normalization of those metrics (distance traveled, and number of cars) a weight function ($f(x)$) is applied to each cluster. We normalize the minimized number of cars with the initial number of cars ($initial_{cars}$), and the minimized traveling distance with the total initial distance ($initial_{distance}$). The result of both weight functions is compared and the best cluster is chosen. The function is

$$f(x) = \frac{(cars_{param} \times (length(x)/initial_{cars})) + (distance_{param} \times (distance(x)/initial_{distance}))}{initial_{distance}}$$

(4) Mapping and Routing: The backend needs to be able to compute routing between users. Mapping and routing is necessary for the implementation of some of our methodologies, but also to obtain the final distances travelled by each group. Such capabilities are available through Application Program Interfaces (APIs).

Google Maps was the API of choice to because of some of its advantages, such as its pervasiveness, spread-out usage, familiarity and its calculation capacities and flexibilities. The API however, has a paid usage model so its use must be minimized.

Database. We chose to store data with a document-oriented NoSQL data model. Go-Together’s data lends itself more to NoSQL than SQL for several reasons:

- Non relational data: Data used across the platform is disjoint and non relatable.
- Unstructured documents: NoSQL allows document flexibility. Events require flexibility as information changes during planning.
- Hierarchical data: Our platform has hierarchical information, best modeled in a NoSQL database.

In it, data is stored in documents, which are organized in collections. Each document contains a set of key-value pairs. Documents can contain sub-collections and nested objects, both of which can include primitive fields like strings or complex objects like lists. The data model must follow the following rules:
As a ride-sharing platform, Go-Together faces the challenge of clustering users. With a focus on collective events, we need to solve n-to-n matchmaking. By analysing all participants locations, event destination and number of cars (and seats) it computes the best possible grouping of participants. The final result is a solution designed to minimize the total distance travelled by the participants and the number of cars.

To solve this clustering problem we studied several methodologies:

- **Combinatorial optimizations**: Specialized approaches designed to obtain optimal results.
- **Heuristics**: General search rules that match participants by approximation in order to reduce computations.

Because one of the objectives of our algorithms is to minimize the total distance travelled, we must first define how it is calculated. We studied three approaches of calculating distance: 1) Euclidean distance; 2) Haversine formula; 3) Real road length.

- **Euclidean distance**: In mathematics, the euclidean distance is the “ordinary” straight-line distance between two points in Euclidean space. Using the latitude and longitude of two participants, we can calculate the straight-line distance between them.
- **Haversine formula**: The haversine formula determines the great-circle distance between two points on a sphere given their longitudes and latitudes. Using it we can get the straight-line distance between two participants considering the curvature of the Earth.
- **Real road length**: Regarding routing problems, traveling distance between two points does not equal to its euclidean distance. Roads are winding and sometimes there’s the issue of one-way streets. To use the true distance participants travel, we request it from an API.

For a focus on optimization and efficiency, euclidean distances are the more appropriate choice. When compared to the haversine formula, there is not much gain (if any) in considering the earth’s curvature and the euclidean formula is more computationally economic. Sometimes however there is the necessity to use road distances, even when considering the round-trip time and monetary usage cost.

**Combinatorial Optimization.** With combinatorial optimization we seek to find the best solution to a problem out of a very large set of possible solutions. In most cases, problems like these have a vast number of possible solutions — too many for a computer to search them all. So, we need to employ a solver to narrow down the search set, in order to find an optimal (or close to optimal) solution.

- **Integer Linear Programming**: The linear optimizer finds the optimal value of a linear objective function, given a set of constraints modeled as linear inequalities. Like all optimization problems, our problem has 2 elements: 1) The objective is the quantity we want to optimize. A function must be defined to calculate the value of the objective for any possible solution. This is called the objective function. 2) The constraints are restrictions on the set of possible solutions, based on the specific requirements of the problem.

It is important to note that an ILP solution verifies the objective function by calculating every possible combination of its constraints.

- **Vehicle Routing Problems**: In vehicle routing problems the goal is to find the most efficient paths to transport items through a complex network. The network is usually represented by a graph where nodes represent locations, arcs represent courses between them, and a route is a path through a set of nodes. Each arc has a weight, corresponding to the cost (distance) of traveling that route. The problem: find a set of paths in the graph (corresponding to delivery routes for each vehicle) that includes every destination while minimizing the total cost.

- **Bin Packing Problem**: The goal in these problems is to minimize the number of bins required to transport all the existing items. We adapted to it with the bins being the available cars and the items the participants. To solve the problem the last requirement is the volume of the bins. In our model the volume of each bin is its corresponding number of seats.

**Heuristics.** The heuristic approaches are designed to reduce computational costs and execution time. They search for feasible configurations using general searching rules to provide a result close to optimal for most use cases. Each heuristic computation consists of two phases:

- (1) Group riders (participants with no car) with drivers (participants with car).
Group drivers together to create the final clusters, consisting of drivers and their riders.

We studied three heuristics:

- **Route shared**: An attempt to minimize deviations, by grouping participants that share the largest percentage of their routes to the destination.
- **Voronoi Cells**: A more greedy approach where participants are clustered with other participants that are closer, creating Voronoi Cells.
- **Radius**: A matchmaking search with radial exploration. Compared with the previous heuristic, instead of attempting to match with a single, closer participant, this heuristic possibilities are narrowed to several participants within its radial search.

### 4 IMPLEMENTATION

In this section, we describe the implementation of our frontend and backend. In particular, how we authenticate users, what information is stored in the database and how we implemented the clustering methods.

#### Mobile

There are several core differences between providing a website or an application. Given the context and necessities of our solution, developing a mobile application makes the most sense. It is an endeavour based on mobility and travelling. And thus, it benefits greatly from offline functionality (to see planned routes) paired with access to intrinsic device functionalities (mostly Global Positioning System (GPS)).

Choosing the right mobile Operating System (OS) for development is a crucial point. There are many differences between different OSs, some more impactful than others. The decision will impact all stages of development, and some common comparisons are market share, profitability, development process and targeted audience. However, the application developed was only meant as a proof of concept and we chose Android as the mobile platform for development. This means functionality is only available through Android devices.

The application contains 6 activities: the **Main Activity**, the **Sign-In Activity**, the **Create Event Activity**, the **Join Event Activity**, the **Event Activity** and the **Update Event Activity**.

The **Main Activity** is the first activity users encounter. It is a login module that allows users to log-in/sign-in using their credentials. We allow users to register using their Google credentials or by providing an email-username-password combination. With either method, the email and password serve as unique access keys, and the username is used a non unique identifier of the user in events within the application. Since data is not device-dependent and is stored in our database, login to our platform allows users to access it on multiple devices.

The **Main Activity** provides an overview of the user’s events and system action. It presents a list of interactable events where the user can perform some actions upon the event or upon its participation in the event, depending on the user credentials. The event itself also displays some information, more specifically, the name of the event, the destination and the current number of participants. In this main screen an expandable button offers the user the option of creating an event or joining an event created by someone else.

Users create their events and define its characteristics in the **Create Event Activity**. The must fill in 4 fields, the desired event’s title, the desired event’s destination, the user’s start location where and if he volunteers to be a driver (with how many seats are available if he does). The location fields make use of an autocomplete feature that makes use of Google’s ‘Places API’. In the the user is prompted with the event’s unique identifier.

Users access the **Join Event Activity** by providing an event’s unique identifier. Then, the screen allows the user to fill-in his own constraints, starting location, if he is a driver and with how many seats if he is.

In the **Event Activity** users can see a detailed visualization of the event. An interactive map shows the destination and starting location, and a bottom slider shows the details of each participant’s participation. For completed events, the map shows the user’s route, and the bottom slider also shows information of the calculated clusters. The option bar allows the user to perform the same event management options he could on the **Main Activity**.

The last activity is the **Update Event Activity**. The activity works similar to the create event activity although aesthetically different at launch with input fields prefilled with the old information.

#### Database

We chose Google’s Cloud Firestore ([25]) to implement our database which allowed easy and direct integration with our Android implementation. The captivating capabilities that made us choose Cloud Firestore were offline support, real-time updates and cloud-hosting.

Our database stores two different collections, one for the users and one for the events, following the rules detailed in Section 3. Individual documents are stored inside those collections with the necessary fields.

**User documents**’ fields are: 1) username, user’s preferred username. 2) events, a list of events the user is partaking in.

**Event documents**’ fields are: 1) completed, a field indicating if the event has been finalized and the clustering calculated.
cluster, a field mapping drivers to the list of their passengers. 3) destination, the destination’s address and coordinates. 4) drivers, a list of all participant capable of driving. 5) owner, the creator of the event. 6) participants, number of participants. 7) title, the event’s title. 8) participants subcollection, with participants’ starting location, if they volunteer as driver and with how many seats.

Authentication
Knowing a user’s identity allows us to securely save user data in the cloud and provide personalized, persistent experience to each user.

The Firebase service provides an authentication system called Firebase Authentication. Firebase Authentication integrates tightly with other Firebase services (like our Firestore database), and it leverages industry standards like OAuth 2.0\(^2\) and OpenID Connect\(^3\). It also provides a complete, ready-to-use drop-in authentication solution called FirebaseUI that handles the UI flows for users signing in with any of the above mentioned credentials. In Go-Together we allow sign in/sign up with e-mail+password and Google Accounts. Although for backend functionalities purposes both credentials are the same, for the user it provides more flexibility. In most cases Android users have their Google account OAuth token stored, meaning they easily can sign in through a single User Interface (UI) prompt.

Microservices

We decided to implement our backend through webservices\(^4\) with microservices\(^5\) that can be requested by the user, using the mobile application. The microservice architecture is, among other things, highly maintainable and testable, loosely coupled and independently deployable.

This allowed us to continuously update and individually test each clustering method without interruption of availability and functionality in the rest of the platform.

We chose Google Cloud Functions\(^{[26]}\) to deploy our microservices, an event-driven serverless compute platform. Cloud Functions allows us to build and deploy services at the level of a single function, scaling automatically in accordance to services’ traffic.

Cloud functions are executed as a response to triggers. These triggers can be events happening within the platform itself (e.g. creation or deletion of database documents) or direct invocation through HTTP requests. JavaScript Object Notation (JSON) is used to interchange data through HTTP requests. Each clustering methodology was deployed as an independent microservice (Figure 2) implemented in Python because of its data types, collection types and abundance of libraries for scientific computing, data analytics and math-intensive operations.

To provide the best, most efficient solution, the individual methodologies were implemented as separate microservices in order to be tested and evaluated (results in section 5).

Microservices can also manage the backend (Figure 2). More specifically, one of our deployed microservices manages the database and the deletion of events. Because deletions are shallow, this deletion-management microservice is triggered every time an event is deleted to remove subcollections and references to the deleted event.

\(\textit{Combinatorial Optimization.}\) Google OR-Tools\(^{[27][28]}\) is an open source software suite for optimization, that provides solvers tuned for tackling problems in vehicle routing, flows, integer and linear programming, and constraint programming.

By default, OR-Tools solves Mixed Integer Linear Programnings (MILPs) using Coin-or branch and cut (CBC)\(^{[29]}\), an open-source solver.

The optimization elements of our problem are: 1) The \textbf{objective:} In our problem, the objective is to minimize distance. As such, in our implementation the objective function calculates the total distance of any assignment of drivers and riders. 2) The \textbf{constraints:} In Go-Together, the only constraint is for each participant to be selected exactly once.

---

\(^2\)OAuth is an open standard for access delegation, commonly used as a way for Internet users to grant websites or applications access to their information on other websites but without giving them the passwords.

\(^3\)OpenID Connect is a simple identity layer on top of the OAuth 2.0 protocol, which allows computing clients to verify the identity of an end-user based on the authentication performed by an authorization server, as well as to obtain basic profile information about the end-user in an interoperable and REST-like manner.

\(^4\)A webservice is a technology for providing services over “web” or HTTP.

\(^5\)A microservice is a software architecture defined by designing software applications as suites of independently deployable services (which can be deployed as webservises).
Consider all possible ways to choose a combination $S$ of 1 driver and up to the number of seats in his car. For each such, compute the total distance $d_S$ of the shortest route that involves that driver picking up all riders and taking them to the destination (a single driver with 5 riders requires $5! = 120$ shortest-path computations). All combinations from an empty car to a full 5 seat car require 154 computations. Then, we introduced a zero-or-one variable $x_S$ for each combination $S$, where $x_S = 1$ means we select that combination. We require that each participant is selected exactly once; thus, for each participant $p$, we add the equality $\sum_{S'} x_S = 1$, where $S'$ ranges over all combinations where $p \in S$. Then, we minimize the linear objective $\sum_{S} d_S x_S$, where $S$ ranges over all combinations. This ensures we minimize the total distance, while ensuring every participant reaches the destination.

In our solution this combinatorial expansion happens when calculating the distance travelled by each participant assignment.

Google’s OR-Tools mitigates the time expansion it takes to solve large VRP problems by sometimes returning solutions that are good, but not optimal. OR-Tools uses an open-source solver developed by Google\cite{28}.

The data required by the OR-Tools VRP solver consists of:

- **Distance matrix:** An array of distances between locations.
- **Vehicles:** The number of vehicles in the fleet.
- **Demands:** Each location has a demand corresponding to the quantity to be picked up.
- **Capacities:** Each vehicle has a maximum quantity that the vehicle can hold. As a vehicle travels along its route, the total quantity of the items it is carrying can never exceed its capacity. Capacities model the empty seats in the vehicle.
- **Starts:** The starting locations of the vehicles.
- **Ends:** The ending locations of the vehicles. In our problem the end is always on the destination.

To build the distance matrix we used Google’s Distance Matrix API. This way we get real distances between participants.

Go-Together needs the drivers to be considered as possible pick-ups for other drivers. To solve this problem, we made two adjustments: 1) We added the drivers’ locations to the list of pick-up points and 2) The drivers themselves have a demand of 0, but their pick-up locations have a demand of 1.

This way, drivers are considered for pick-ups. If the participant is selected to drive, then it will pick himself up because the distance to himself is 0, otherwise his location will be contained in another driver’s cluster.

The bin packing implementation was achieved with the use of a python module called ‘binpacking’\cite{30}. At the time of writing the module uses a Best-Fit-Decreasing (BFD) heuristic. However, the bin packing problem solves problems with bins of constant volume. In our paradigm bins (cars) have different volumes (seats), so we needed to make some adaptations. The solution we developed was:

1. Sort cars by capacity (i.e. seats) first (riders are not considered). Items’ weights are the size of the different groups of passengers (1 for riders and drivers alike).
2. Run the algorithm with bins of capacity equal to that of the car with more available seats.
3. Among the bin allocations produced by the algorithm, choose the bin with more items in it (meaning more groups of passengers were able to fit inside). In case of several bins, choose the one with less distance travelled.
4. Remove the grouped cars (the bin + the items) from the sorted list and from the unplaced items.
5. Re-run the bin packing algorithm with the next emptiest car and with the remaining passenger groups.
6. Repeat until no more packing is possible.

The reason we sorted by descending number of seats is trivial to understand. Given car $A$ and $B$. Consider that car $A$ has a higher capacity than car $B$, and consider a possible solution (smallest number of cars) containing $B$ as a bin, but not $A$. If we swap cars $A$ and $B$, we will not need more cars, and because $A$ has more empty seats, we may even fit the passengers from another car. So, which cars are used in an optimal solution is obvious. We use cars sorted by capacity in descending order.

**Heuristics.** To compare participants, before clustering them together, we defined the notion of Matchmaking Quality Heuristic (MQH) as a value that classifies the quality of clustering together two participants (based on the heuristic). The comparison of the MQH is different for each heuristic. The best quality may be the highest or lowest available MQH. The MQH is used to cluster users in two phases (Algorithm 1).

For the first phase, the MQH is calculated for each rider-driver pair. Then, rider’s MQHs are compared to match them with best available driver. Available, because if the driver already has a full car, the next best driver is compared until a match is achieved.

For the second phase, the MQH is calculated for every pair of drivers. Then, just like in the first phase, drivers are matched with their best available MQH. This time, the receiving car needs to have enough available seats to transport the old driver plus its riders. There is no guarantee of a merge between cars. In the worst case, all cars go unmatched and thus need to drive to the destination.
Route Shared. This heuristic is calculated using Google’s Directions API, which returns detailed information for each leg of the requested route. The routes requested re from participants’ starting location to the destination.

For the first phase of the computation, riders’ route are compared with drivers’ route and ordered by the amount shared. The amount shared equals the number of shared nodes divided by the total nodes in the driver’s route, to return how much of the driver’s route is shared with the interested rider. Grouping is then done in descending order, from most route shared to the least.

The second phase is just like the 1st phase except this time we compare drivers’ route.

In the second phase of the heuristic, comparisons could be more correct if the routes included the riders. However, that choice requires more API calls. More precisely, one for each created group in phase one, and one every time a match was found in phase two (to update the routes for subsequent comparisons).

Voronoi cells. This heuristic matches participants that are closest together. To compare distances we used the Euclidean formula, it is the least computationally expensive calculation and it is faster because it is done locally. Grouping is then done in ascending order, from closest to furthest.

In traffic networks sometimes the closest distance ’in a straight line’ is not the closest distance by car. Resorting to the API we could use driving distances, which also factor in altitude. But, calculation time would increase as it would take into account round-trip delay times, as well as use (riders * drivers) + (drivers² − drivers) calls.

Radius. For this heuristic, the MQHs are calculated in incremental radial steps. Participants are sorted in ascending order by radial step.

For the first phase, we used an increment of 5 Km. The center of the calculation is the rider’s pickup location. For drivers inside the same radius range no particular order was considered.

For the second phase, an increment of 10 Km was used. Like in the first phase, no particular order was taken for possible drivers inside the same radius.

\[ f = \left(\frac{\text{distance} + (\text{radius\_step} – 1)}{\text{radius\_step}}\right) \cdot \text{radius\_step} \]

We used the above formula, where \(\text{radius\_step}\) is the aforementioned incremental values and \text{distance} is the distance between participants (as in the previous heuristic, same considerations for calculation apply). This formula returns the closest multiple of \(\text{radius\_step}\) rounded up, while direct multiples stay the same.

Verifying cumulative distances. Since the goal of the heuristic approaches is to minimize the total distance travelled, after a suitable match is found, the algorithm can verify if the cumulative distance is shorter with the cars grouped or separated. This may not guarantee an optimal solution (if there is two possible shorter groupings \(A\) and \(B\), but we choose \(A\) first and \(B\) is even shorter) but it guarantees that cars are only grouped if the distance is factually reduced.

However, this assertion was not considered because of its cost. To verify the cumulative distance it is necessary to calculate individual and joint routes, which can be made in two ways:

1. Calculating locally using any measure of distance: Every possible permutation of riders needs to be calculated before choosing the shortest one. Considering the worst case scenario, where drivers only match in the last comparison it is a considerable number. Expensive for larger events.
2. Using Google’s API to obtain distance values: The expense comes in API requests. The API request can be specified to internally optimize the route by rearranging the waypoints in the most efficient order. This makes it so that each verification requires only three requests. But, also assuming the worst case scenario the number of API calls can be prohibitively expensive for large events.

Ordering. After the calculations of the MQHs, there is an ordering concern for the matchmaking iterations. For example, are riders matched with drivers, or are drivers matched

---

**Algorithm 1:** High-level description of the two phased heuristic clustering.

```plaintext
/* 1st Phase */
foreach rider in riders do
  foreach driver in drivers do
    MQH[driver][rider] ←− calculate_MQH(rider, driver)
  end
  group_with_best_match(rider, MQH)
end
/* 2nd phase */
foreach driver in drivers do
  remaining_drivers ←− drivers − driver
  foreach new_driver in remaining_drivers do
    MQH[driver][new_driver] ←− calculate_MQH(driver, new_driver)
  end
  group_with_best_match(driver, MQH)
end
```

Radius. For this heuristic, the MQHs are calculated in incremental radial steps. Participants are sorted in ascending order by radial step.

For the first phase, we used an increment of 5 Km. The center of the calculation is the rider’s pickup location. For drivers inside the same radius range no particular order was considered.

For the second phase, an increment of 10 Km was used. Like in the first phase, no particular order was taken for possible drivers inside the same radius.

\[ f = \left(\frac{\text{distance} + (\text{radius\_step} – 1)}{\text{radius\_step}}\right) \cdot \text{radius\_step} \]

We used the above formula, where \(\text{radius\_step}\) is the aforementioned incremental values and \text{distance} is the distance between participants (as in the previous heuristic, same considerations for calculation apply). This formula returns the closest multiple of \(\text{radius\_step}\) rounded up, while direct multiples stay the same.

Verifying cumulative distances. Since the goal of the heuristic approaches is to minimize the total distance travelled, after a suitable match is found, the algorithm can verify if the cumulative distance is shorter with the cars grouped or separated. This may not guarantee an optimal solution (if there is two possible shorter groupings \(A\) and \(B\), but we choose \(A\) first and \(B\) is even shorter) but it guarantees that cars are only grouped if the distance is factually reduced.

However, this assertion was not considered because of its cost. To verify the cumulative distance it is necessary to calculate individual and joint routes, which can be made in two ways:

1. Calculating locally using any measure of distance: Every possible permutation of riders needs to be calculated before choosing the shortest one. Considering the worst case scenario, where drivers only match in the last comparison it is a considerable number. Expensive for larger events.
2. Using Google’s API to obtain distance values: The expense comes in API requests. The API request can be specified to internally optimize the route by rearranging the waypoints in the most efficient order. This makes it so that each verification requires only three requests. But, also assuming the worst case scenario the number of API calls can be prohibitively expensive for large events.

Ordering. After the calculations of the MQHs, there is an ordering concern for the matchmaking iterations. For example, are riders matched with drivers, or are drivers matched
```
with riders? Also, in what order are they considered, is it random? In our implementation, the order of the iterations is the order the participants are retrieved from the database fetch.

To not increase overhead computations no other order was implemented. Regardless, it would be possible to order the iterations based on:

- Number of seats: We can start by filling in the one with most empty seats, allowing smaller cars to group with bigger cars. The idea is to minimize the number of cars used.
- Distance to destination: Group with the closer cars first. The idea is to minimize distance travelled by preventing backtracks. Further correctness could be added by determining the heading of the route (a participant south of the destination should not pick-up a participant North of it).

5 EVALUATION

In order to find out which of the implemented methodologies is the best, we need to evaluate them. We wanted to answer two questions with our evaluation:

(1) What is the quality of the solution provided by each implemented methodology?

(2) How efficient is each implemented methodology?

We want to assert the quality of the calculated cluster regarding its total travelling distance and the number of cars (both should be as low as possible). Furthermore, we want to evaluate each method’s performance in being capable of providing an acceptable response time.

Each clustering method was individually tested under different circumstances and several metrics were retrieved. Each test was ran with all clustering methods to compare their quality and performance.

Experimental Methodology

All methodologies were ran as a deployed microservices in Google Cloud Functions (GCF). Each microservice was deployed with an allocated memory of 256MB and a timeout of 60 seconds, with the only exception being the ILP implementation which has 2GB of allocated memory and a timeout of 540 seconds.

We tested the quality and performance of our methodology with volume testing and tests of specific topographic dispersions of participants. Each test is a designed event, and, like a regular event, they were saved to the database and the microservice triggered through its HTTP handler.

During the execution of the algorithm some metrics were logged, so that they could be extracted and used for comparison. For each execution we logged the execution time, the total distance travelled and its standard deviation between cars, the final number of cars, standard deviation of occupied seats and the standard deviation of vacant seats. Although, we devised solutions to minimize the total distance and number of cars, the standard deviation metrics could provide us insight into which solution provided more homogeneous results, a possible factor for users’ satisfaction.

Volume testing. For volume testing we devised twenty events each one with more participants than the previous. The first test had 5 participants, and the last one had 50 participants. From 5 participants to 50, the incremental step was the same for all twenty events ((5 – 50)/20). The intention was to evaluate the algorithms’ capabilities when subjected to increasing volumes of data.

To provide authenticity and diversity with undocorred constraints the parameters were randomized for each test. Every participants’ location and the destination is different between tests. Locations were kept to the center of Portugal, with the latitude limited from 39.213133ºN to 39.676679ºN and the longitude from 8.830733ºW to 8.390575ºW.

However, driver-to-rider ratios where kept consistent. This way we could compare results and answer our evaluation’s questions under different ratios. We tested with \( \frac{1}{4} \), \( \frac{2}{4} \) and \( \frac{3}{4} \) ratios.

Topographic dispersion. To test the flexibility of our methodologies, we came up with four different topographic dispersions that we felt would encompass most types of events in our platform.

(1) Participants near each other, but with a destination far from them.

(2) Participants distant from each other and from the destination.

(3) Participants near each other, with the destination in the center of them.

(4) Participants distant from each other, with the destination in the center of them.

When referring to participants near or distant from each other, the idea was to remove them from each others vicinities because some of the implemented algorithms work based on proximity.

When defining the destination, the idea was to make sure all cars eventually had the same route when far away, and that cars would have mirrored routes when in the center. This is because some algorithms work based on routing, and a center location forces them to consider any possible backtrack.

Each topographic dispersion was tested five times with increasing participant numbers, and event’s constraints were kept as identical as possible between tests. Under each dispersion the driver-to-rider ratio was always \( \frac{1}{2} \), the destination
the same, and participants starting positions where kept for subsequent tests.

Results
The ILP approach was unable to compute any tests of medium size events or higher due to memory expansion (and being forcefully terminated by GCFs).

The results regarding the standard deviation of cars’ vacancy and occupancy were inconclusive. The results had too much noise and we were unable to conclude anything.

Execution time is higher when there are more drivers to consider. On events of smaller scale, all methodologies provide an efficient computation. The VRP solution had overall higher execution times. Notwithstanding, the radius heuristic had the best performance.

The VRP consistently excels in computing clusters with less total distance, providing the higher quality solution. In smaller events there is minimal difference between approaches. For larger events the only comparable results were those of the voronoi heuristic.

None of the approaches were able to provide consistent homogenisation between distances travelled by cluster’s cars. Its a hard metric to achieve in parallel unless it’s being specifically optimized to provide equal values between cars. The bin packing approach consistently computes clusters with fewer cars, separating itself from other approaches even in smaller events. The radius heuristic deserves mentioning especially at higher ratios, where it closely compares to the bin packing approach, where it mostly computes equal solutions with the occasional solution with one extra car.

For the range and amount of tests performed we saw no major impact from any type of topographic dispersion. Like the results of volume testing, the VRP was consistently the best at optimizing the distance travelled and the bin packing solution at minimizing the number of cars. The bin packing heuristic underperformed with Participants near each other and the destination far from them. This could be because of poor performance of the BFD heuristic in this particular topographic dispersion.

Results Discussion
The ILP is absent from the majority of the test cases because it required a lot of memory, capping the 2GB of memory allocated for the microservice. The tests it completed, it was able to compute clusters with minimal distances. Its execution time however was considerable.

The methodology with higher quality solutions when minimizing distances is the VRP. However, for small execution times the Voronoi heuristic distinguishes itself. The VRP approach did not always compute the cluster with the least travelled distance. This is due to how the VRP computes its solutions, creating an approximate first solution before searching for new ones. Sometimes to prevent the execution times seen with the ILP approach, the iteration stops before finding the optimal solution.

The bin packing method did not always compute the cluster with the least amount of cars. Although those cases are minimal, it can be explained by the BFD heuristic used by the module since heuristics only provide approximations not guaranteeing optimal results. The bin packing methodology should be used in lower driver-to-rider ratios, and the radius heuristic in higher radius (given that results where comparable but execution times were smaller for the heuristic).

6 CONCLUSIONS
We were successful in developing a platform capable of clustering users in an n-to-n paradigm form ride-sharing. The ride-sharing events were created with an Android mobile application that served as the front-end of our platform for users to fill-in their constraints before clustering them in our back-end. It was scalable through as microservices, with a single public facing microservice in charge of the previous computation flow.

Our clustering algorithm tackled two problems: minimize total distance traveled by users and minimize the number of travelling cars. The solution has to provide the best possible results for our two problem while being efficient and scalable.

To provide the best solution we implemented several different clustering approaches: three heuristics and three combinatorial optimization solutions. We then ran several tests to compare their performance and efficiency. The evaluation showed that each had individual strengths that became particularly apparent in large events, or larger rider-to-driver ratios.

In conclusion, to provide the best, most efficient cluster possible (in regards to the discussed minimization choices) the back-end needs to compute two clusters, one for each parameter, before comparing them through a weight function. We suggest some overhead analyses before invoking the appropriate approach.

• Minimizing distance is a choice of execution time vs quality. For a faster solution one should employ the Voronoi heuristic. For quality, the VRP methodology provides the best results.
• To minimize the number of cars it requires an analysis on the driver-to-rider ratio, employing a bin packing solution for lower ratios and the radius heuristic for higher ratios.
REFERENCES


