Algorithms and Data Structures for Efficient Ride Sharing Platforms

Francesco Tosoni, Paolo Ferragina, Andrea Marino, Giovanni Resta, Paolo Santi

***Abstract*—Ride sharing has been recently shown to have a tremendous potential to reduce the number of vehicles needed to serve a certain mobility demand, such as one based on taxis [1], [2]. However, although car pooling services have developed in recent years and are now widely available worldwide (e.g. Uber, Didi, Lyft), ride sharing techniques still suffer severe scalability limitations, especially if the goal is combining multiple on-demand ride requests into a single trip in a large urban area. As a result, any on-demand mobility system has to eventually address the following specific problem: given a number of ride requests and available vehicles, how to efficiently and effectively estimate travel times between ride request endpoints and vehicle positions so to provide to the following trip construction and routing steps only a relevant subset of them. Brute force approaches based on considering all possible combinations of request endpoints and vehicles are not scalable for real-time computation.**

In this paper we present a novel locality filtering approach that, combined with a scalable ride sharing algorithm based on shareability networks, is able to substantially speed up known approaches while only minimally impacting the quality of the computed ride sharing solution. We corroborate this novel proposal with a large set of experiments, for which we have utilized a dataset consisting of one month of trip requests ($\sim 10^6$) performed in two different urban areas, namely Manhattan (NYC) and Singapore.

***Index Terms*—Intelligent transportation systems, ride sharing, vehicle routing, urban mobility, locality filtering, geometric data structures.**

I. Introduction

RIDE SHARING (or carpooling) represents a long-standing proposition for decreasing road traffic while satisfying the need of mobility for a population of travelers living in a well-defined geographical area. Many ride sharing services are available nowadays (e.g. Uber, Didi, Lyft, just to mention some of them), and the impact of ride sharing techniques is expected to become even more important in the future with the introduction of self-driving vehicles.

F. Tosoni is with the Department of Computer Science, Università di Pisa, 56127 Pisa, Italy, as well as with the TeCIP Institute, Sant'Anna School of Advanced Studies, 56124 Pisa, Italy. (e-mail: f.tosoni@studenti.unipi.it).

P. Ferragina is with the Department of Computer Science, Università di Pisa, 56127 Pisa, Italy (e-mail: paolo.ferragina@unipi.it).

A. Marino is with the Department of Statistics, Informatics and Applications, Università di Firenze, 50134 Firenze, Italy (e-mail: andrea.marino@di.unifi.it).

G. Resta is with Istituto Informatica e Telematica (IIT), Consiglio Nazionale delle Ricerche, 56100 Pisa, Italy (e-mail: giovanni.resta@iit.cnr.it).

P. Santi is with Istituto Informatica e Telematica (IIT), 56100 Pisa, Italy (e-mail: paolo.santi@iit.cnr.it), and with MIT Senseable City Lab, Cambridge, MA 02139, USA.

This work has been supported in part by the 2019 MIT-UniPi Grant Program.

Manuscript received *****; revised *****.

Recent work has shown that in dense urban areas – such as Manhattan (NYC) – it is in principle possible to combine most point-to-point ride requests (e.g., taxi rides), and hence achieve a significant reduction in the number of vehicles needed to serve them and in the total kilometers traveled by the taxi fleet [1], [2], [3].

Unfortunately, the application of these approaches to real-time scenarios still poses significant scalability challenges. In fact, an on-demand ride sharing service needs to collect three types of input data in order to attempt to optimize the combination of taxi requests into a single ride to be eventually assigned to a vehicle. One type of input is made of ride requests, that are typically collected from customers through a (customer) smartphone app. Ride requests define, among other parameters, the starting point of the trip and the desired destination, which we call *trip endpoints* in the following. A second type of input is the position and status of the vehicles in the fleet, which is also acquired through a (driver) smartphone app. The third type of input needed to optimize the mobility service is an accurate estimate of travel times between (a subset of) trip endpoints and (a subset of) vehicles in the fleet. This last type of input is key to the optimization process as it allows to accurately estimate the efficiency of different combinations of trip requests into shared rides, and their assignment to available vehicles. The smaller is the subset of such *candidate matching trips*, the faster is the solution to the optimization problem, but the worse *could be* the final ride sharing solution computed by the optimization algorithm both in terms of total travelled kilometers, waiting time of the travellers, or number of used vehicles.

So far, research on scalable solutions for on-demand shared mobility has focused mainly on the efficacy of the optimization process itself, by assuming that its input of ride requests, vehicle positions, travel time estimations and corresponding candidate matching trips is given. This is the case, for instance, of the approaches introduced in [1], [2], [3]. However, while providing ride requests and vehicle positions is relatively straightforward to implement, this is not the case for the selection of a relevant subset of candidate matching trips. Clearly, a brute-force approach that computes the travel times for all possible combinations of trip endpoints and vehicle positions is not feasible, both in terms of computational time (that would be quadratic in the number of trip requests $\mathcal{T}$), space occupancy and cost. In fact, each trip comparison results in a few (possibly expensive) shortest path calculations [4], [5], [6], [7], which may lead to evaluate distances even between trip pairs whose endpoints appear very far apart each other, and which will most probably not result in a feasible match.

Even in the case that travel time estimates are obtained from commercial services, like Google Traffic, these charge costs based on the number of submitted queries, which could be $\Theta(|\mathcal{T}|^2)$ in the naïve approach. Thus, smarter solutions must be found to efficiently filter, and then compute, the travel time of only an appropriate subset of candidate matching trips without looking to all their (quadratic) pair combinations, and without loosing the ones that contribute to the computation of the optimal ride sharing solution.

To address this challenge, we introduce in this paper a novel filtering approach based on spatial and geometric considerations which is aimed at substantially reducing the number of travel time estimates, and thus candidate matching trips, needed as input to the subsequent optimization process. While the proposed filtering approach is general enough to be applied to a variety of other optimization problems in the context of on-demand mobility, in this paper we show its specific application to the problem of efficiently and effectively approximating the optimal set of matching pairs of trips in a taxi ride sharing application.

A. Our contribution

We present a novel algorithm and related data structures which efficiently solve the ride sharing problem.

Our algorithm improves the one proposed in [1], via a proper orchestration of geometry-based conditions with time-based conditions that allow to achieve a stronger pruning effect on the number σ of candidate trips to be checked for matching. We leverage empirical statistical metrics of the city graph to introduce a filtering technique, able to define a locality area in which it is more likely to find candidates for a combination with a given trip request. The locality filter, which is a key building block in our solution, can be adapted to deal with different speed networks and traffic conditions. Furthermore, it is able to *self-tune* itself when we consider heterogeneous areas of the city (e.g. periphery and downtown), where it is common to experience variations in the average speed.

In the following we will refer to the original approach [1] as the *legacy* algorithm which incurs in the following two main limitations:

- It lacks any spatio-temporal correlation, so that the feasible match between two trips is verified even if these two trips refer to locations within the city that are too far from each other. For this reason, the number of pairs (T_i, T_j) to be checked might grow *quadratically* in the number of trips i.e. $\Theta(|\mathcal{T}|^2)$;
- It makes massive use of the expensive Dijkstra's shortest-path algorithm in order to evaluate the distance of each pair of nodes (i.e. origins, destinations) in the city graph.

Our novel algorithm, instead, provides the following three-fold contribution:

- It reduces the number of pairs (T_i, T_j) to be checked by applying a set of *conditions of geometric proximity* between trips T_i and T_j.
- It reduces the time for the retrieval of the pairs (T_i, T_j) which appear *geometrically close*, by leveraging some provably efficient geometric data structures for spatial searches.
- It deploys state-of-the-art algorithms for distance calculation, by leveraging the planar structure of the city graph and its embedded (and sometimes hidden) highways. As a consequence, it achieves dramatically improved shortest-path calculations.

As a result, our filtering approach builds efficiently in both time and space a *pruned* version of the *shareability network* introduced in [1], that allows to optimally match ride requests and thus compute an optimal ride sharing solution in real time. We will then evaluate the efficiency and effectiveness of our filtering approach by comparing the optimality of the ride sharing solution derived by our pruned network against the optimal one obtained by considering a *complete* network in which the travel times of all possible pairs of matching trips are evaluated [1].

We will corroborate the asymptotic analysis of our algorithms with a large set of experiments conducted on two datasets of taxi hailings issued in 2011 and referring to two urban areas of different magnitude: the small district of Manhattan in New York City ($59.1\ km^2$), and the metropolis of Singapore ($721.5\ km^2$). We have also analyzed the efficiency of our solution over different time slots (e.g. rush time, nighttime). In our experimental investigation we have observed that our approach is able to induce a speed up up to 5X (or even more) over [1], especially during the so-called "rush time", namely when ride sharing systems have to deal with larger numbers of user requests. We are also able to match thousands of trip requests in few seconds, so that the possible delay induced by the completion time of our algorithm can be considered *de facto* negligible, and thus suitable for an application in a ride sharing app.

Overall, the results that we report in this paper clearly show that our solution is robust under different scenarios, and is flexible enough to balance recall (i.e. number of retrieved feasible candidates) and time speed up, as well as to be extended to the problem of the vehicle-to-trip association, thus giving suggestions for the development of similar algorithms and data structures in this context too.

II. Related work

App-based on-demand ride services (also known as "ride-sourcing" services) are transforming urban mobility by providing timely and convenient transportation to anybody, anywhere and anytime. As observed in [8], independent research on the use of ridesourcing is very limited; nevertheless there is a rich literature on the related topic of ridesharing (carpooling), and on taxis fleet management in particular.

Several independent studies conducted in different urban areas across the globe (i.e. Hangzhou [9], 62 cities of Japan [10], New York [1], [11], San Francisco [8], Shenzen [12], Singapore [13], Zurich [14]) provided several insights into the expected usage characteristics of ridesharing or ridesourcing, and their potential economical and social impacts.

In particular, Santi *at al* [1] were the first to propose a novel graph-based method to evaluate the impact of the *sharing*

economies. Their work almost surprisingly highlighted that up to 94.5% (or even more) taxi trips in Manhattan can be shared by two taxi users, if a small tolerated delay (e.g. one minute) is introduced. More, recently, [11] proposed a real-time dynamic trip sharing algorithm based on the request-trip-vehicle shareability graph (RTV-graph). This algorithm starts from a greedy trip-vehicle assignment, and improves it through a constrained optimization process which converges over time to the optimal trip-to-vehicle matching.

However, a common limitation of the approaches by [1] and [11] is that they need to pre-compute and store in advance an all-to-all shortest-path matrix consisting of a travel-time lookup table between any source-destination pair within the city road map. This matrix requires a space occupancy that grows *quadratically* with the size of the city roadmap (i.e. number of crossroads), thus posing severe scalability issues.

The literature offers a very rich variety of shortest-path algorithms [4], [5], [6]. Generally, these solutions are based on some pre-computed data structure, which is able to speed-up the query time distance computation. As expected, the solutions which pre-compute and store *larger* data structures exhibit better performance at query time, so that it is up to the user to choose the algorithm which offers the best space-time tradeoff according to her application needs.

In many cases, the achieved speed-up is obtained by directing or pruning the classical Dijkstra shortest-path algorithm. In APPENDIX A we review some of the most efficient and effective approaches to this issue.

Given the observations in the APPENDIX A, we will leverage the CH technique as a distance-computation module for our ridesharing approach. Nevertheless, it goes without saying that, any state-of-the-art techniques can be used to solve the shortest-path queries because our approach uses them as a black-box.

III. RIDE SHARING MODEL

We model the demand for taxi service in a certain geographical area (e.g., a city) as a set $\mathcal{T}$ of taxi requests issued by potential customers. Each trip/ride $T_i \in \mathcal{T}$ is defined by (1) the GPS coordinates of its origin o_i and destination d_i, (2) and by its starting time st_i. Note that the arrival time at_i is estimated as $at_i = st_i + tt_i(o_i, d_i)$, where $tt_i(o_i, d_i)$ is the estimated time to go from the origin o_i to the destination d_i.

The road network of the geographical region in which travelers can issue requests for the taxi service is modeled by means of a *city graph* G_A, where the set of nodes V_A represents crossroads, that are assumed to be the only valid origins and destinations for T_i, and the set of (weighted) edges E_A provides estimates of the time needed to go across the corresponding road segments. The weights on the edges of the graph G_A are time-varying according to different traffic patterns during different hours of the day.

We borrow from [1] a set of constraints, to be fulfilled:

- A *shareability parameter* s, limiting the amount of trips that can be "combined" together: namely, the maximum number of customers that can share the same vehicle.
- A *quality of service parameter* Δ, limiting the delay that passengers can incur due to ride sharing. A high value of Δ may combine more trips, at the price of an increasing delay (discomfort) for passengers to arrive to their final destination. The value of Δ should be determined by the traffic managers of a city or the mobility operator, and it is suggested to be 5 minutes in dense urban environments as that considered in [1].
- A *time window parameter* δ, which is used to batch a number of trip requests for improving the quality of the final solution. This parameter represents also an upper bound to the time of a (customer) taxi app to return a potential matched $(ride, taxi)$ assignment to the customer, and thus the availability of the shared trip. Parameter δ has the effect of reducing the candidate rides to be matched to the ones satisfying the condition $|st_i - st_j| \leq \delta$.

Given the tuple $(G_A, \mathcal{T}, s, \Delta, \delta)$, the *ride sharing problem* consists of finding a feasible grouping $\mathcal{M}$ (also known as *matching*) for the trips in $\mathcal{T}$ such that (1) each trip is combined with at most $s - 1$ other trips of $\mathcal{T}$, (2) each customer experiences at most Δ minutes of delay in getting to its destination (w.r.t. the time the customer would have spent by traveling alone), (3) the starting time of every trip is at most δ minutes after its request.

The authors of [1] proposed two different models for the computation of the matching $\mathcal{M}$ from the set of trips $\mathcal{T}$:

- the *Offline model* (aka oracle model) represents an omniscient and artificial scenario in which trip-sharing decisions can be taken by considering not only the current taxi requests, but also all the future ones (hence $\delta = +\infty$), thus offering a theoretical upper bound for sharing opportunities;
- the *Online model* represents a realistic scenario in which a customer, using an "e-hailing" application, issues a taxi request by reporting pickup and drop-off locations, and within the small time window δ receives feedback from the taxi management system on whether and which shared ride is available for her.

In this paper we will analyze both scenarios, with particular focus on the Online model which is at the core of real-time mobility platforms.

IV. THE SHAREABILITY NETWORK

The shareability network concept has been introduced in [1] to quantitatively assess the benefits of large-scale ride sharing on sustainability. The shareability network translates spatio-temporal sharing problems into a graph-theoretic framework where, namely, a solution for the ride sharing problem translates into the classical problem of finding a maximum matching in a graph. The vast potential of this framework has been confirmed by the significant traffic reduction in the borough of Manhattan [1].

Following [1], we will limit our analysis to the case in which a trip can be combined with *at most* one other trip: hence, $s = 2$. This limitation has been introduced by the authors of [1] for computational reasons. In fact, the case $s > 2$ induces the shareability network to have a hypergraph structure, with up to s nodes connected via a link, and the cost of finding a

maximum matching in that shareability hypergraph turns out to be NP-hard [15]. Even though setting $s = 2$ could imply that taxi-sharing services, and social sharing applications in general, will likely be able to combine only a limited number of trips, it has been shown that the proposed approach is able to provide immense benefits to a dense enough community like the area of Manhattan in New York [1].

For the concerns of the ride sharing problem over a given set $\mathcal{T}$ of trips as defined in SECTION I), the *shareability network* G_{SN} associated with $\mathcal{T}$ is an unweighted non-directed graph whose nodes represent the trips T_i in $\mathcal{T}$, namely $V_{SN} = \{T_i \mid 1 \leq i \leq n\}$, and two nodes T_i and T_j are connected through an edge $(T_i, T_j) \in E_{SN}$ *iff* it results that $|st_i - st_j| \leq \delta$ *and* that the two trips T_i and T_j can be served by the same taxi ensuring the quality of service parameter Δ. When we consider the *Online model*, we also add the constraint that two taxi requests cannot be matched with each other if the difference of their starting times is greater than δ.

In this framework, any *matching* in G_{SN}, i.e. a set of edges not having common vertices, corresponds to a feasible solution for the ride sharing problem, as it translates in feasible pairings of trips in $\mathcal{T}$. In particular, a *maximum matching* corresponds to the combination of trips of the largest size while respecting the delay requirements of the ride sharing problem. Surprisingly, it has been shown in [1] that, in densely enough urban areas such as Manhattan (NYC), it is possible to combine 92% of trips even when considering $s = 2$ and small values for the tolerated delays (i.e. $\delta = 1$ minute and $\Delta = 5$ minutes).

Unfortunately, populating the shareability network turns out to be very expensive if it is approached in a brute-force manner. In fact, we cannot afford neither $\Theta(n^2)$ Dijkstra-like computations, nor we can precompute and store (for serving subsequent travel-time queries), the all-pairs-shortest-path matrix (e.g. using Floyd Warshall algorithm, or repeated Dijkstra computations) in $\Theta(n^2)$ space.

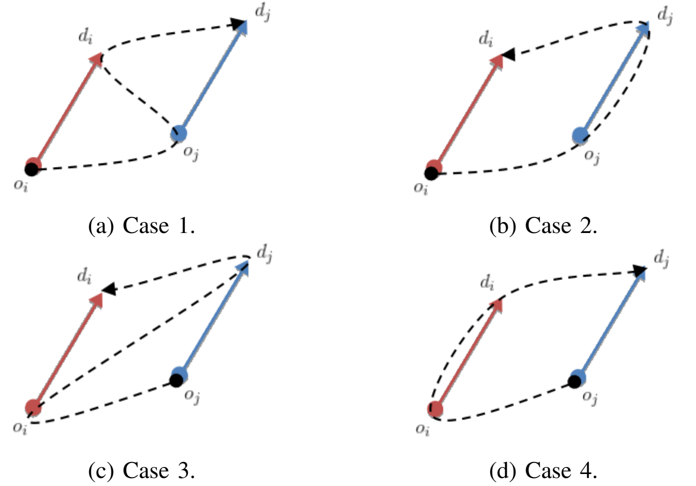
In order to make the computation of G_{SN} practically affordable, we need to reduce the number of shortest-path queries issued on the graph G_A by selecting a proper subset of pairs of trips to be checked for candidate matches. On the other hand, working on a subset of the original shareability network, the computed solution (i.e. maximum matching) might have lower quality (i.e. number of edges) than the original one. The rest of the paper is therefore devoted to exploring this tradeoff between computational/storage efficiency and quality of the computed solution.

A. Necessary and sufficient conditions

In order to achieve the reduction of shortest path queries, it is useful to reason in terms of necessary time conditions that have to be met in order to match two trip requests with each other, and hence form a combined trip.

By assuming that $s = 2$, the authors of [1] observed that the combination of two trips T_i and T_j can occur in 4 different configurations, as shown in FIGURE 1.

A first way to combine two trips T_i and T_j is to collect (in order) the corresponding passengers and deliver them in the same order to their respective destinations. We call this kind of match, *match of the first kind*. Alternatively, one can collect (in order) the passengers of trips T_i and T_j, and then deliver to destination first the passenger of trip T_j, and then the one of trip T_i. We call this kind of match, *match of the second kind*. There are two other (symmetric) cases to be considered, that are obtained by inverting the roles of T_i and T_j in the two cases described above; these symmetric cases correspond to scheduling first the pick-up for trip T_j rather than the pick-up for trip T_i.



(a) Case 1. (b) Case 2. (c) Case 3. (d) Case 4.

Fig. 1: Valid ways to combine a pair of trips. The red and blue lines represent resp. trips T_i and T_j, whereas the black dashed lines represent the 4 different paths that the vehicle can take in order to combine T_i and T_j. Cases 1 and 3 are called *match of the first kind*, because passengers are collected and delivered in the same order; otherwise, we call them *match of the second kind*.

For matches of the first kind, the resulting trip corresponds to the path $o_i \rightarrow o_j \rightarrow d_i \rightarrow d_j$. This occurs when the following equations, denoted hereafter as $A(i, j)$, are satisfied.

$$\begin{cases} st_j \leq st_i + tt(o_i, o_j) \leq st_j + \Delta & (1) \\ st_i + tt(o_i, o_j) + tt(o_j, d_i) \leq at_i + \Delta & (2) \\ st_i + tt(o_i, o_j) + tt(o_j, d_i) + tt(d_i, d_j) \leq at_j + \Delta & (3) \end{cases}$$

Eq. 1 is needed to ensure that the second customer (of trip T_j) is picked with a delay no longer than Δ time after his scheduled starting time (otherwise the delay of the combined trip would surely be larger than Δ). Eq. 2 (resp. Eq. 3) guarantees that the customer of trip T_i (resp. of trip T_j) is delivered to the destination with a delay no larger than Δ.

For matches of the second kind, the resulting trip corresponds to the path $o_i \rightarrow o_j \rightarrow d_j \rightarrow d_i$, and can be obtained from the matches of the first kind by simply inverting the delivery order of the two passengers. Similarly to the previous combination, this case occurs whenever the following system of inequalities, denoted hereafter as $B(i, j)$, is satisfied:

$$\begin{cases} st_j \leq st_i + tt(o_i, o_j) \leq st_j + \Delta & (4) \\ st_i + tt(o_i, o_j) + tt(o_j, d_j) + tt(d_j, d_i) \leq at_i + \Delta & (5) \end{cases}$$

Eq. 4 ensures that the customer of trip T_j is picked up no later than Δ time after her scheduled starting time (otherwise the delay of the combined trip would surely be larger than Δ). On the other hand, Eq. 5 guarantees that the first customer (i.e. the customer of trip T_i) is served within the permitted delay parameter Δ.

It is worth noting that in this case there is no need to explicitly check that the second customer (of trip T_j) is served within Δ seconds of delay, because trip T_j is served without intermediate steps, and thus the taxi follows the fastest route for it. Hence, all that we need to check for trip T_j is just that the pick-up operation is scheduled within a reasonable time, as stated in condition 4.

As already observed in [1], and because of symmetry, conditions for the case $o_j \to o_i \to d_j \to d_i$ are given by $A(j,i)$; and conditions for the case $o_j \to o_i \to d_i \to d_j$ are given by $B(j,i)$. In summary, we have:

Lemma 4.1: Two trips T_i and T_j can be matched if, and only if, $|st_i - st_j| \le \Delta$ and the following holds

$$A(i,j) \vee B(i,j) \vee A(j,i) \vee B(j,i).$$

V. Locality filtering

All of the matching conditions presented in the previous section are exclusively *time* based. Here we introduce some novel *geometric*-based conditions for a match between two trips which exploit some statistical properties of the city graph. Our intuition is clearly stated in the following claim, which we first prove empirically.

Claim 5.1: There exists a strong correlation between: (1) the traversal time of a source-to-destination path (shortly, s-d path), and (2) the s-d Euclidean distance.

In order to experimentally verify Claim 5.1 we considered the set $\mathcal{P} = \mathcal{P}(\Delta_S)$ of all possible paths in the city graph G_A which can be traversed in less than Δ_S. Let us denote with $d(p)$ the distance covered by a path $p \in \mathcal{P}$. We also denote by $F_{\mathcal{P}}(\cdot)$ the empirical cdf of the distribution of the distance values l, i.e.:

$$F_{\mathcal{P}}(l) = \frac{|\{p \in \mathcal{P}|\ d(p) \le l\}|}{|\mathcal{P}|} \tag{6}$$

Let us denote with $d(\mathcal{P})$ the support of the empirical distribution of path distances, namely $d(\mathcal{P}) = \{d(p) \mid p \in \mathcal{P}\}$.

We are interested in the distance value l_{95} such that $f(l_{95}) = 0.95$, thus corresponding to the *95-percentile* value of the $d(\mathcal{P})$ distribution. The value l_{95} can be interpreted as a statistically-grounded upper bound to the Euclidean distance that a taxi can traverse in that city graph by using a traveling time Δ_S.

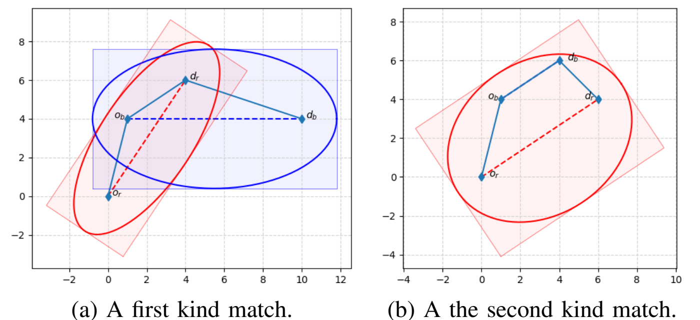
For the computation of distance d we used in our experiments the *Euclidean distance* between the points in the Cartesian plane corresponding to the origin and destination of the evaluated path. Given the curvature of earth's surface, a known alternative to distance estimation is the *Haversine formula*, which finds application e.g. in the calculation of ocean routes, and it is often used within very wide geographical areas. However, the geographic scale of our study is small (i.e. a city), so that the use of the Euclidean distance as a close approximation of geodesic distance is justified.

Our experimental spatio-temporal analysis is synthesized in Appendix B where it is evident that time and distance values exhibit a strong correlation, which motivates us to approximate distance values starting from time values via a e.g. linear regression approach. We hence introduce the symbol γ to denote the time-to-distance mapping which assigns to each time value Δ_S a corresponding (bound on the) travelled distance l_{95}: i.e., $\gamma(\Delta_S) = l_{95}$. In the experimental section, we will investigate experimentally how the change in the percentile, i.e. from l_{95} to l_{40}, will change the speed up versus recall of our proposed solution.

Repeating the percentile preprocessing for different time values Δ_S (e.g. multiples of 5 minutes) we collected enough samples which are then interpolated to define γ-mappings for every continuous time value. We have also specialized the mapping γ for different districts within the city graph. This specialization turns out to be particular effective to capture the complexity and heterogeneous nature of speed networks, especially for maps from the old world (i.e. Europe, Asia), where there can be substantial differences in registered vehicle speeds in various city areas (i.e. downtown versus periphery). For details, see the Appendix to the original paper [1].

A. From time conditions to necessary space conditions

Leveraging the time-to-distance mapping γ and Claim 5.1, we can thus reformulate the time-based conditions for trip matches of Section IV into geometric-based conditions, as follows.



(a) A first kind match. (b) A the second kind match.

Fig. 2: Combined trips of the two different kinds: subscripts R and B refer to trips T_R and T_B, respectively, which are surrounded by ellipses red and blue. The colored rectangles wrapping the ellipses will be introduced in Section VI.

Let us start by considering the matches of the *first kind*, and refer to Fig. 2 (a) where the trips T_i and T_j are indicated with dashed lines and the solid line is used to denote the combined trip T_{ij}. Without loss of generality, we restrict ourselves to the case in which the origin of T_{ij} coincides with the origin o_i of the trip T_i. In the symmetric case where the combined trip T_{ij} has its origin in o_j rather than in o_i, we can follow similar considerations to derive conditions $A(j,i)$.

The set of conditions $A(i,j)$ introduced in Section IV correspond to the (necessary and sufficient) conditions for the realization of a match of the first kind. Let $D_i = \gamma(\Delta)$ be the the distance value associated to the tolerance parameter Δ of Section I, derived by means of the time-to-distance

mapping γ. Notice that D_i depends on source i whereas Δ does not depend on i; this is a direct consequence of the fact that the time-to-distance mapping γ has been specialized for different districts, and hence the distance D_i associated to Δ is not uniform within the city graph G_A. Furthermore, let $l_i = \gamma(tt(o_i, d_i))$ be the distance associated with the expected service time $tt(o_i, d_i)$ of trip T_i. The parameters D_j and l_j can be defined in a similar manner.

Because of the definition of γ-mapping, we can reformulate the equations of $A(i, j)$ in geometric queries specified in the following lemma, proved in Appendix C. Here the reformulation is "approximate" because the γ-mapping reflects the time-to-distance relation up to the 95% of the paths in our city graphs. In the experimental section we will evaluate the accuracy of this approximation over our dataset of city graphs referring to Manhattan and Singapore.

Lemma 5.2: We can translate the feasibility condition $A(i, j)$ of a matching of the first kind into the (approximately equivalent) geometric-based system of inequalities:

$$\begin{cases} d(o_i, o_j) + d(o_j, d_i) \leq l_i + D_i & (7) \\ d(o_j, d_i) + d(d_i, d_j) \leq l_j + D_j & (8) \end{cases}$$

We can provide a geometric interpretation of the first (resp. the second) inequality above: it corresponds to the membership of the node/point o_j (resp. d_i) to an *ellipse* which wraps entirely the trip T_i (resp. T_j) – see Figure 2-A.

We now focus on the matches of the second kind. Again, without loss of generality, we restrict ourselves to the analysis of the case in which the combined trip T_{ij} includes in its entirety the optimal (i.e. shortest) path for the trip T_j (the case of T_{ij} including the shortest-path for T_i boils down to analogous considerations). The feasibility of a combined match of the second kind is determined by the two conditions defined in $B(i, j)$ (see SECTION IV), which can be rephrased again as the two geometric queries specified in the following lemma, proved in Appendix C.

Lemma 5.3: We can translate the feasibility condition $B(i, j)$ of a matching of the second kind into the (approximately equivalent) geometric-based system of inequalities:

$$\begin{cases} d(o_i, o_j) + d(o_j, d_i) \leq l_i + D_i & (9) \\ d(o_i, d_j) + d(d_j, d_i) \leq l_i + D_i & (10) \end{cases}$$

This system of inequalities corresponds geometrically to the membership of the nodes/points o_j and d_j into an *ellipse* which wraps entirely trip T_i – see Figure 2-B. The ellipse for this second kind match is the very same ellipse that we discussed for first kind matching cases.

VI. OUR PROPOSAL

We will see in this section how the geometric-based filtering techniques presented in SECTION V can be used to implement an efficient and efficacious ride sharing algorithm.

Using Lemmas 5.2 and 5.3, we can efficiently find a set of *candidate* trips for feasible matches satisfying the conditions $A(i, j)$, $B(i, j)$, $A(j, i)$, $B(j, i)$. They are retrieved by executing ellipse-based range searches over the Cartesian plane. Since our geometric conditions are necessary but not sufficient to guarantee the feasibility of the match between two trips, these range searches will find some *false positive* matches, so that we need to execute a post-check over the retrieved set of candidate trips. We will show that these false positives are not many in practice and thus the overall resulting algorithm will show effective performance.

More specifically, our algorithm proceeds as follows. We construct two posting lists $OT[i]$ and $DT[i]$ for each trip T_i in the input set $\mathcal{T}$.

- $OT[i]$ is defined as the set of trips whose *origin* belongs to the ellipse of trip T_i.
- $DT[i]$ is the set of trips whose *destination* belongs to the ellipse of T_i.

For DT we also build the inverse posting list DT^{-1} such that, for every pair of trips $(T_i, T_j) \in \mathcal{T}^2$ it results that: $i \in DT^{-1}[j]$ if, and only if, $j \in DT[i]$.

The key algorithmic observation here is that we can turn the retrieval of candidates trips for a match with trip T_i into the execution of some posting-list intersections. In particular, according to the geometric interpretation of LEMMA 5.2 in SECTION V, we can obtain the candidate set for a first kind match by computing $OT[i] \cap DT^{-1}[i]$. Similarly, according to the geometric interpretation of LEMMA 5.3 in SECTION V, the candidate set for a second kind match can be obtained by computing $OT[i] \cap DT[i]$.

APPENDIX D contains the pseudocode and an exhaustive explanation of the algorithmic procedures which compute the candidate sets of feasibile matches of either of the two kinds.

VII. FULL CHECK OF THE FEASIBLE MATCHES

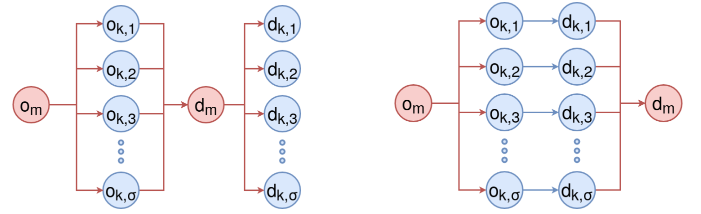
We are left with the description of the algorithmic twists we introduce to speed up the computation of the one-to-many and many-to-many shortest paths involved in the implementation of the procedures to check the candidates found in the previous phase (denoted as `check_candidates_1k` and `check_candidates_2k` in APPENDIX D).

This check could be very expensive if approached by means of a brute-force processing of the underlying city graph (as e.g. the classic Dijkstra's algorithm) because each of the queries involved in those checks could spur a certain number of traversal calculations in the city graph.

The literature offers also several graph reduction (lossy compression) techniques for shortest-path distance estimation within a (quasi-)planar graph like the one of a road network (see e.g. [4], [5], [6]). The proposed techniques are based on several heuristics that compute some auxiliary data, such as additional edges (shortcuts) and labels or values associated with vertices or edges, which are then used to accelerate an arbitrary number of shortest path queries, typically by pruning or directing Dijkstra's algorithm. Heuristics within this framework are based on a wide variety of ideas, such as arc flags, landmarks, (contracted) highway hierarchies and transit nodes, just to mention a few (see [6] and refs therein). Recently, Akiba *et al.* proposed in [16] an algorithm, called *pruned highway labeling* (PHL), that achieves provably good results in terms of both query and pre-processing time. Nevertheless, a main limitation of the distance estimation algorithm of [16]

SOURCE	DEST.	$A(i,j)$	$B(i,j)$	$A(j,i)$	$B(j,i)$
o_i	d_i	✓	✓	✓	✓
o_j	d_j	✓	✓	✓	✓
o_i	o_j	✓	✗	✓	✗
o_j	o_i	✗	✓	✗	✓
d_i	d_j	✓	✗	✗	✓
d_j	d_i	✗	✓	✓	✗
o_i	d_j	✗	✓	✗	✗
o_j	d_i	✓	✗	✗	✗

TABLE I: Shortest-path queries executed to assess A()s and B()s conditions between the source and the destination of the involved pair of trips. The needed shortest-path computations are marked with a *tick* (✓); those that are not needed, are instead marked with a cross (✗).



(a) Path computations for first kind matches.

(b) Path computations for second kind matches.

Fig. 3: Shortest path calculations needed to find candidates for: (A) the condition $A(m,k)$ (feasible match of the first kind), and (B) the condition $B(m,k)$ (feasible match of the second kind). Here, we have denoted as usual with T_m the trip that we process during the m-th iteration of ALGORITHM 2. In sub-figure (A), all the needed traversal times (red arrows) can be evaluated using one-to-many and many-to-one distance computation techniques. In sub-figure (B) – instead – the red arrows correspond to traversal times which can be computed by means of many-to-many distance computations, whilst the σ blue arrows *incoming* represent instead σ different one-to-one Dijkstra-like path calculations. These one-to-one traversal times correspond to estimating, for each of the σ involved trips, the time needed to serve the corresponding passenger when considered in isolation and by using one single taxi.

is that it deals exclusively with *undirected graphs*, which are not able to capture the presence of one-way streets, as well as the variation of average speed that a taxi user can experience while traversing the same road segment in either of the two directions.

In our experiments we have used the *Contraction Hierarchies* (CH) technique described in [17] for two main reasons:

- it is able to treat *directed* graphs;
- the API offers the possibility to perform optimized one-to-many and many-to-one shortest path calculations.

One should also notice that the shortest path calculation of our approach can be looked at as a *black box*, so many other alternative solutions could be plugged in our code without changing its algorithmic structure.

In order to understand which is the role played by the one-to-many and many-to-one shortest path calculations, one can look to TABLE I which lists the shortest-path queries needed to assess the conditions $A(i,j)$, $B(i,j)$, $A(j,i)$ and $B(j,i)$. The pattern of shortest path calculations in TABLE I are derived from FIGURE 2 and also pictorially represented in FIGURES 3 (A) and (B). In particular, we notice that first we need to evaluate the traversal times from the origin to the destination of each trip (first two rows of the table, or blue arrows in FIGURE 3 (B)). These times can be pre-computed and stored in a global vector, so that they can be easily retrieved each time they are needed for subsequent checks. Then, we need to evaluate a shortest path query between a location (origin or destination) of one trip T_i and a location (origin or destination) of another trip T_j (as indicated in all the other rows of TABLE I). If we imagine to pin T_i and take T_j among all trips resulting from the intersections $OT[i] \cap DT[i]$ and $OT[i] \cap DT^{-1}[i]$, then we are turning several one-to-one path queries, into a single one-to-may path query. This is the key observation underlying the speed up introduced for testing candidates.

VIII. EXPERIMENTAL EVALUATION

This section is devoted to show the results obtained by applying out approach to two real datasets of taxi requests issued in February-March 2011 in the borough of Manhattan (NYC) and Singapore. The dataset of Manhattan has been utilized also by the authors of [1], and consists of a subset of a larger dataset which comprises 172 million trips served by 13,586 taxi cabs in New York during the calendar year 2011. For Singapore we have utilized a dataset with roughly 11 million taxi requests issued between the 14th February and 14th of March 2011, which has been used in [18].

We compare the results of our algorithm with the results achieved by the legacy solution in [1]. Both algorithms have been implemented in c++. They have been compiled with the g++ compiler of the *GNU* suite using the option $-O3$. We have then executed the compiled code on a commodity laptop equipped with an i5-8250 CPU and 8GB of DDR4 RAM Memory. All experiments ran in internal memory.

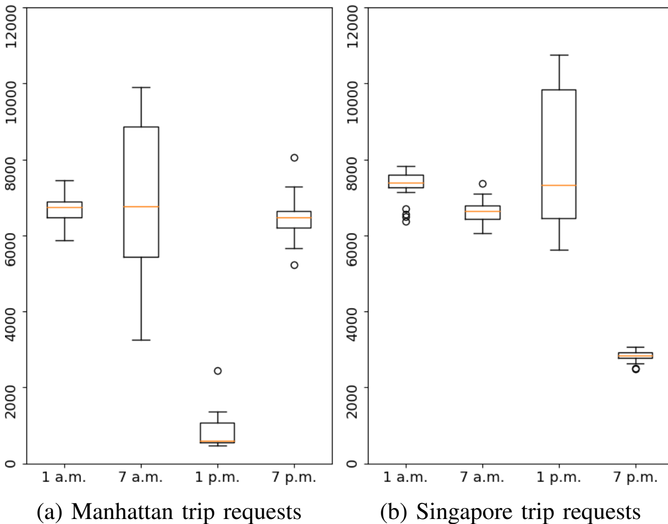
a) Dataset Description: For both Manhattan and Singapore we have considered trip requests issued in the time span between 14th February 2011 1 p.m. (UTC) and 14th March 2011 1 p.m. (UTC). We have considered several time windows in order to test the performance of our solution under different scenarios (e.g. rush time, nighttime). In particular, we have extracted those requests which have been performed in the time intervals: [1.00–1.20 a.m.], [7.00–7.20 a.m.], [1.00–1.20 p.m.], [7.00–7.20 p.m.]. For the experiments we have used the parameter values $\Delta = \delta = 5$ minutes.

For both Manhattan and Singapore we estimated the average time speed of the cabs during different time slots using the methods presented in the Appendix of [1]. For the estimation of the average time speed in Singapore we also differentiated between: weekdays (wd), Saturdays (sat), and Sundays (sun). TABLE II reports some dataset metrics, whereas FIGURES 4(a) and (b) show the variation in the number of trip requests as a function of the issuing hour. It is of great importance to correctly estimate the traffic congestion in terms of number of taxi requests issued within a certain time span; this because vehicle traffic has a serious impact on the

average time speed and hence on the estimated traversal times within locations of the city.

DATASET	TRIP REQUESTS	CROSSROADS (NODES)	ROADS (EDGES)	WD SAT/SUN
manhattan	581 631	4 091	9 452	✗
singapore	688 383	11 789	26 223	✓

TABLE II: Dataset description. In the *trip requests* column we report the number of requests in the extracted portion of the dataset.



(a) Manhattan trip requests (b) Singapore trip requests

Fig. 4: Box-and-whisker plots reporting the number of trip requests issued in (a) Manhattan and (b) Singapore around the four time-slots considered. The data refer to a 20 minutes batch of requests.

b) Performance Measures: We will denote as R the number of taxi requests. Let us denote by N and L, respectively, our new algorithm and the legacy algorithm. Given an algorithm $A \in \{N, L\}$, let us consider the following quantities:

- S_A be the number of shortest path queries performed by A,
- T_A be the completion time of algorithm A (including the construction of the shareability network and its matching phase),
- F_A be the number of feasible matches (that is, the number of edges in the shareability network computed by A). We have $F_N \subseteq F_L$.
- M_A be the size of the maximum matching found by A, which equals the number of matched trip pairs.

By making use of this notation, in order to study the performance of N and L, we define the following measures.

- The *matching size reduction*, that is the reduction in the number of matched trip pairs (expressed in percentage): $(M_L - M_N)/M_L$.
- The *speed up factor* achieved by N wrt L, as T_L/T_N.
- The *feasible matches reduction* (expressed in percentage): $(F_L - F_N)/F_L$.
- The *reduction of the evaluated distances*, that is reduction in the number of issued shortest-path queries (expressed in percentage): $(S_L - S_N)/S_L$.

Furthermore, we will denote with γ_{40} (resp. γ_{95} the time-to-distance mapping built by using 40-percentile (resp. 95-percentile) distance values. The γ_{40} and γ_{95} settings are represented in FIGURES 5 and 6 through empty and filled dots, respectively.

c) Performance plots: FIGURES 5(a) and 6(a) show the matching size reduction as a function of the number of requests, resp. for Manhattan and for Singapore. Namely for each experiment considering x number of requests and $y = (M_L - M_N)/M_L$ reduction of matched trips, we draw a circle in position (x, y). Clearly, the reduction is desired to be as small as possible, meaning that the matching achieved by N is not much smaller compared to the one found by L. Interestingly, FIGURES 5(a) and 6(a) show that the gap in the number of matched trips between N and L decreases as the number of trip requests increases (i.e. the matching size reduction becomes negligible). This result is verified under all the considered scenarios.

Different colors represent varying time slots, in particular: red stands for 1 a.m., orange stands for 7 a.m., blue stands for 1 p.m., and green stands for 7 p.m. In the case of Manhattan, the blue and orange cases perform better with γ_{95} rather than with γ_{40}. Instead, for the green and red distribution ("rush time") we register no significant difference in the figures when using either of the two time-to-distance mappings. Furthermore, for green and red distributions the matching size obtained by L and N can be considered the same. In the case of Singapore, for the blue, orange, and green distributions, we also observe that γ_{95} is strictly better than γ_{40}, while in the red distribution, the matching size does not substantially change when using either of the two algorithms or time-to-distance mappings.

Overall, if we look at the γ_{95} distributions in Manhattan (FIG. 5(a)), only at 1 p.m. we register a (very small) reduction in the matching size (about 10% on average); for the other time slots no significant reduction is actually observed, as the reduction is consistently negligible. On the other hand, looking at the γ_{95} distributions in Singapore (FIG. 6(a)), we register a reduction of about 15% on average only for the 7 p.m. distribution; for the other time slots no significant reduction is actually observed. The reduction in matching size corresponds to the time slots with a *smaller number of taxi requests*, and hence to the scenarios in which there are fewer feasible candidates for a given match (see FIGURES 4, 5 (A), 6 (A)). In those cases, the shareability network is rather sparse, so that a further pruning can induce some (indeed small) reduction in the number of combined trips. When considering γ_{40} distributions in cases with fewer trip requests (for instance, looking at Figure 4, 1 p.m. for Manhattan and 7 p.m. for Singapore), this sparsity is even more extreme and the matching found by N can be considerably smaller than the one found by L (see respectively blue and green distributions in 5 (a), and 6 (a)). But in those (verifiable) cases the N algorithm could automatically turn to the L algorithm, thus preserving the speed and accuracy because of a sparse graph G_{SN}.

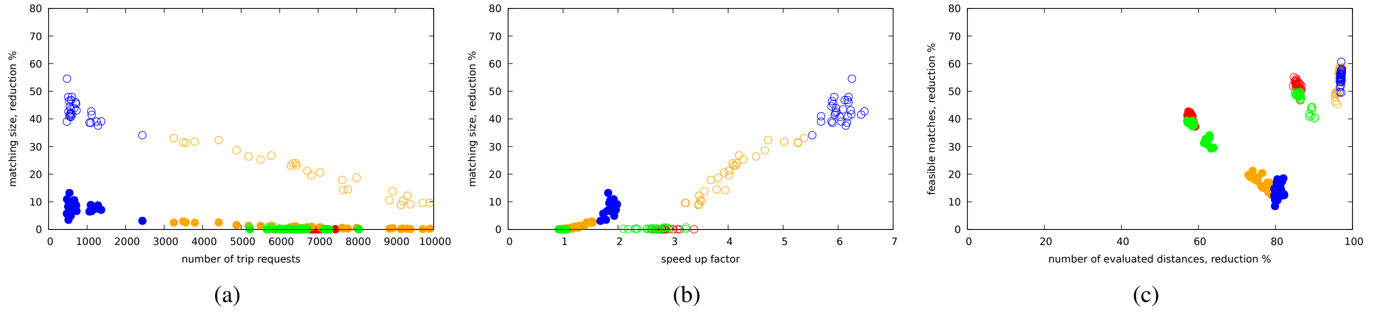


Fig. 5: Performance metrics collected for *Manhattan* (NYC). Filled dots (●) represents metrics obtained using γ_{95} (*95-percentile* distance values), whilst empty dots (○) represent distance values obtained using γ_{40} (*50-percentile* distance values). Different colors represent varying time slots, in particular: red (●,○) stands for 1 a.m., orange (●,○) stands for 7 a.m., blue (●,○) stands for 1 p.m., green (●,○) stands for 7 p.m.

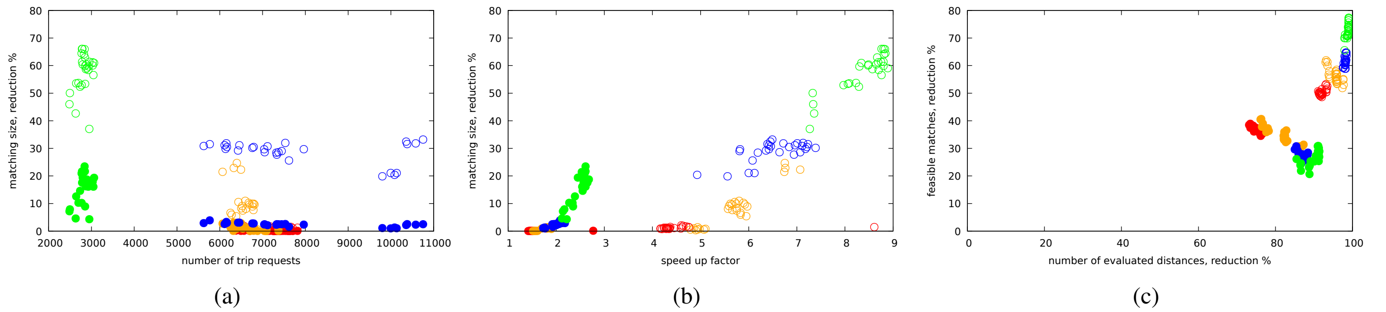


Fig. 6: Performance metrics collected for *Singapore*. Conventions as in FIGURE 5.

Vice versa, it is interesting to notice that our novel solution performs very well in all the situations of *traffic congestion* (the so-called "rush time") where we can indeed register a win-win situation: (a) the same matching size as in the legacy algorithm, and (b) much faster response time, and especially when compared to the legacy solution which is characterized by long computation latencies, as shown next.

FIGURES 5(b) and 6(b) show the matching size reduction $y = (M_L - M_N)/M_L$ as a function of the achieved time speed up $x = T_L/T_N$ in both Manhattan and Singapore. Clearly, an higher speed up x and a lower matching size reduction y is desired. Looking at FIGURES 5(b) and 6(b), the speed up is very high in the case of γ_{40}, showing an implicit trade-off between the size of the matches found and the speed up factor. However, with a negligible matching size reduction, our N is often up to 3 times faster than L in the case of Manhattan and up to 5 times faster than N in the case of Singapore.

The reason for the achieved speed-up can be immediately clarified looking at FIGURES 5(c) and 6(c). For each experiment we draw a circle at position indicated by the reduction of evaluated distances $x = (S_L - S_N)/S_L$ and the feasible matches reduction $y = (F_L - F_N)/F_L$. As y basically corresponds to the reduction of the size of G_{SN}, we can see that the shareability network produced using γ_{40} is in general smaller than the one produced using γ_{95}. This reduction in the number of feasible matches has a positive impact on time speed up, since a sparsification of the shareability network translates into a faster execution of the final maximum matching algorithm. In any case, both for γ_{40} and γ_{95} our new algorithm reduces the number of distance queries of one or two orders of magnitude.

IX. CONCLUSION

In this paper we have presented a novel *locality filter* which we have leveraged to implement a new and more efficient ride sharing algorithm which improves the state-of-the-art in [1]. Experiments over two city graphs of Manhattan and Singapore showed that our solution is very efficient in practice and robust under various parameter settings and time slots.

The delay incurred by our ride sharing solution is negligible (roughly 1 second every 1000 input trips), especially when compared with the time delay imposed by the parameters δ and Δ, which the authors of [1] called *quality of service* and *time window* (see the introduction). As a result, the delay experienced by a potential customer of our ride sharing system is only dominated by: (1) the waiting for the batch δ, and (2) the extra delay (at most Δ) introduced when being served together with another customer.

We notice that our locality filtering approach could be applied not only to speed-up the trip-to-trip matching (as largely discussed throughout this paper), but also to solve the trip-to-vehicle matching: *aka* the minimum fleet problem [2], which consists of finding the minimum number of taxi cabs to serve a set of taxi requests. In this case, we can again figure out to use a locality filtering technique which: (1) selects for each taxi driver a set of n candidate trip requests (e.g. the closest), (2) populates a shareability network keeping track of opportunities for a trip-to-vehicle match, and (3) finally executes a matching algorithm on top of this shareability network to compute the optimal fleet dispatching scheme.

REFERENCES

[1] P. Santi, G. Resta, M. Szell, S. Sobolevsky, S. H. Strogatz, and C. Ratti, "Quantifying the benefits of vehicle pooling with shareability networks," *Proceedings of the National Academy of Sciences*, vol. 111, no. 37, pp. 13 290–13 294, 2014.

[2] M. Vazifeh, P. Santi, G. Resta, S. Strogatz, and C. Ratti, "Addressing the minimum fleet problem in on-demand urban mobility," *Nature*, vol. 557, 05 2018.

[3] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, and D. Rus, "On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment," *Proceedings of the National Academy of Sciences*, vol. 114, no. 3, pp. 462–467, 2017.

[4] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, "Fast shortest path distance estimation in large networks," in *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, ser. CIKM '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 867–876.

[5] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. Werneck, *Route Planning in Transportation Networks*, 11 2016, vol. 9220, pp. 19–80.

[6] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck, "Highway dimension, shortest paths, and provably efficient algorithms," in *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '10. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2010, pp. 782–793.

[7] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies," *Transportation Science*, vol. 46, no. 3, p. 388–404, Aug. 2012.

[8] L. Rayle, D. Dai, N. Chan, R. Cervero, and S. Shaheen, "Just a better taxi? a survey-based comparison of taxis, transit, and ridesourcing services in san francisco," *Transport Policy*, vol. 45, pp. 168 – 178, 2016.

[9] X. M. Chen, M. Zahiri, and S. Zhang, "Understanding ridesplitting behavior of on-demand ride services: An ensemble learning approach," *Transportation Research Part C: Emerging Technologies*, vol. 76, pp. 51 – 70, 2017.

[10] R. Abe, "Introducing autonomous buses and taxis: Quantifying the potential benefits in japanese transportation systems," *Transportation Research Part A: Policy and Practice*, vol. 126, pp. 94 – 113, 2019.

[11] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, and D. Rus, "On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment," *Proceedings of the National Academy of Sciences*, vol. 114, no. 3, pp. 462–467, 2017.

[12] Y. M. Nie, "How can the taxi industry survive the tide of ridesourcing? evidence from shenzhen, china," *Transportation Research Part C: Emerging Technologies*, vol. 79, pp. 242 – 256, 2017.

[13] Y. Wang, B. Zheng, and E.-P. Lim, "Understanding the effects of taxi ride-sharing — a case study of singapore," *Computers, Environment and Urban Systems*, vol. 69, pp. 124 – 132, 2018.

[14] P. M. Boesch, F. Ciari, and K. W. Axhausen, "Autonomous vehicle fleet sizes required to serve different levels of demand," *Transportation Research Record*, vol. 2542, no. 1, pp. 111–119, 2016.

[15] S. Kushagra, "Three-dimensional matching is np-hard," 2020.

[16] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 349–360.

[17] P. Sanders and D. Schultes, "Highway hierarchies hasten exact shortest path queries," in *Algorithms – ESA 2005*, G. S. Brodal and S. Leonardi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 568–579.

[18] R. Tachet, O. Segarra, P. Santi, G. Resta, M. Szell, S. Sobolevsky, S. H. Strogatz, and C. Ratti, "Scaling law of urban ride sharing," *Nature Scientific Reports*, no. srep42868, 2017.

[19] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes, "In transit to constant time shortest-path queries in road networks," in *ALENEX*. SIAM, 2007.

[20] J. Arz, D. Luxen, and P. Sanders, "Transit node routing reconsidered," in *Experimental Algorithms*, V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 55–66.

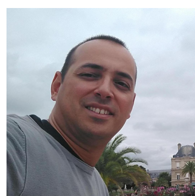
[21] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *Experimental Algorithms*, C. C. McGeoch, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 319–333.

[22] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "A hub-based labeling algorithm for shortest paths in road networks," in *Experimental Algorithms*, P. M. Pardalos and S. Rebennack, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 230–241.

[23] T. Columbus and R. Bauer, "On the complexity of contraction hierarchies," 2009.

Francesco Tosoni received the M.Sc. degree in computer science and networking jointly from the University of Pisa and Sant'Anna School of Advanced Studies in spring 2020. His master's thesis was focused on the development of locality filtering techniques applied to on-demand urban mobility problems. He is currently research assistant at the Acube laboratory directed by professor P. Ferragina. His interests range from algorithms for real world applications to information retrieval and big data.

Paolo Ferragina is Professor of Algorithms at the University of Pisa, where he serves also as Vice-Rector on ICT and President of the PhD in Computer Science. He funded an leads the Acube Lab, where researchers design algorithms for Big Data, mainly in the form of texts and graphs, in collaboration with companies worldwide: Google, Bloomberg, Tiscali, Yahoo!, ST Microelectronics, etc.. His research results got four US Patents and some international awards: e.g. "1995 Best Land Transportation Paper Award" from IEEE Vehicular Technology Society; "1997 EATCS PhD Thesis Award"; "1997 Philip Morris Award on Science and Technology"; and three Google Research Awards. He's serving in the Editor Board of the Journal of Graph Algorithms and Applications (JGAA), he was in the Steering Committee of the European Symposium on Algorithms (ESA), and one of the Area Editors of two Encyclopedias: Algorithms and Big Data Technologies (Springer). He (co-)authored more than 160 (refereed) publications, some books and chapters, achieving an H-index of 31 on Scopus and more than 8500 citations on Google Scholar.



Andrea Marino Assistant Professor at University of Florence. Previously, Assistant Professor at University of Pisa, post-doc at the Laboratory of Web Algorithmics of University of Milan. PhD in Computer Science at University of Florence and best Italian PhD Thesis on algorithms, automata, complexity and game theory 2013 according to the Italian Chapter of the EATCS (European Association for Theoretical Computer Science). Interested on graph algorithms with applications to enumeration, web crawling, bioinformatics, and real-world graph analysis.

Giovanni Resta Giovanni Resta is a Senior Researcher at the Istituto di Informatica e Telematica (IIT-CNR) in Pisa, Italy. His past research interests include computational complexity (especially in relation to linear algebra problems), parallel and distributed computing, and the modeling and analysis of wireless ad hoc networks. In the last few years his research area broadened to include intelligent transportation systems and smart mobility. In particular he is interested in optimization problems related to ride sharing potential and fleet reduction.

Paolo Santi is Research Director at the Istituto di Informatica e Telematica, CNR, Pisa, and affiliated with MIT Senseable City Lab where he leads MIT/Fraunhofer Ambient Mobility initiative. Dr. Santi holds a "Laurea" degree and the PhD in computer science from the University of Pisa, Italy. Dr. Santi is a member of the IEEE Computer Society and has recently been recognized as Distinguished Scientist by the Association for Computing Machinery. His research interest is in the modeling and analysis of complex systems ranging from wireless multi hop networks to sensor and vehicular networks and, more recently, smart mobility and intelligent transportation systems. In these fields, he has contributed more than 140 scientific papers and two books.

APPENDIX A
DISTANCE COMPUTATION TECHNIQUES

The Transit Node Routing (TNR) approach [19], [20] is characterized by almost constant-time queries especially for long paths but, in turns, it results in a much slower (sometimes not scalable) preprocessing phase. A key component of TNR is the so-called *locality filter*, which is able to discriminate between close and far destinations. We will borrow from [19] the term "locality filter" to denote in this paper a completely new spatio-temporal filtering approach. The Contraction Hierarchies (CH) [21] is another well known distance-estimation technique which builds upon the observation that road networks are usually hierarchical. So the speed-up in the shortest path computations is achieved by creating "shortcuts" in a preprocessing phase, which are then used during a shortest-path query either to skip over "unimportant" crossroads, or to save the distance between two important junctions so that the algorithm doesn't have to explore the full path between these junctions at query time. Another approach is called Hub-Labelling method (HL) [22]. A label $L(V)$ (i.e. string containing distance information) is pre-computed in advance for every node v. At query-time the distance between two nodes u and v is estimated looking only at $L(u)$ and $L(v)$ labels. This labelling method supports very fast queries, about five accesses in main memory according to some experimental results [5]. A drawback with respect to other state-of-the-art approaches is the pre-processing time as well as the space occupancy. In fact, some experiments conducted over the "Western Europe" road map [5] have shown that, with respect to CH, the preprocessing phase of HL is roughly 7 times slower and generates 47 times larger data structures.

Lastly, the Pruned Landmark Labeling (PLL) [16] has been recently proposed to exploit the (almost) planar structure of road maps. However, this approach deals with *undirected graphs*, hence it is unsuitable for real road networks.

APPENDIX B
SPATIO-TEMPORAL CORRELATION

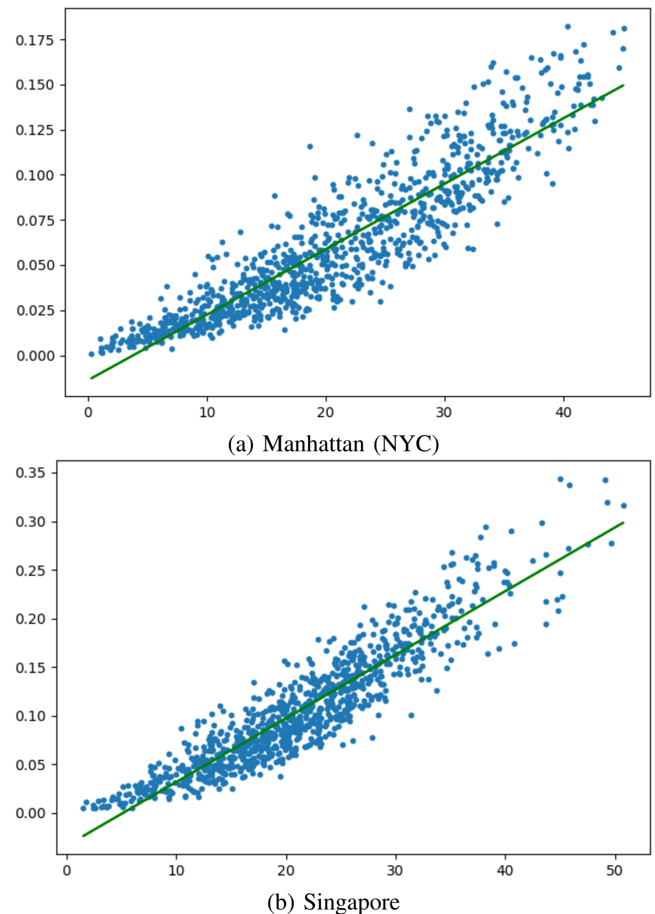
We report herein the result of our experimental analysis described in SECTION V. The scatter plots of FIGURE 7 refer to the cities of (a) Manhattan, and (b) Singapore. The two plots show the strong correlation existing between time and distance values in the analyzed city graphs. This allow us to approximate traversal time values (x-axis) using distance values (y-axis).

During our experiments we have registered a rather large gap between the distance values corresponding to the 100-percentile and the 95-percentile of the distributions in FIGURE 7. This suggests that very few shortest-paths with duration bounded by Δ_S manage to cover large distances. Furthermore, we noticed clustered occurrences of distance values, concentrated in small ranges. This clustering effect further justifies our good approximation of travelling times in cities with Euclidean distances.

As a result of these two observations, we have limited the range of distances covered by trips that travel for less than Δ_S time to the only ones that have destinations not farther than l_{95}. This will discard just the 5% of the valid paths. As in SECTION V we use the symbol γ to denote the time-to-distance mapping. From the linear regression of FIGURE 7 we can derive an approximation for the time-to-distance association γ as:

$$\gamma(\Delta_S) \approx a \cdot 0.95 \cdot \Delta_S + b \tag{11}$$

where a and b represent the slope and the intercept (resp.) of the green line.



(a) Manhattan (NYC)

(b) Singapore

Fig. 7: Plots (a) and (b) report Euclidean distance values (y-axis) as a function of travel time values (x-axis). More in detail, each of the 10^3 blue dots refers to a shortest path between two random locations in (a) Manhattan, and (b) Singapore. Traversal times are measured in minutes. Distance values are computed as Euclidean distances using *global positioning system* (GPS) coordinates, and hence are pure numbers. It is straightforward to note that time and distance values exhibit a strong correlation, which motivates us to approximate distance values (y-axis) starting from time values (x-values) via a e.g. linear regression approach. Indeed, the green line in the plots represent a linear approximation of $d = d(\tau)$ as $a \cdot \tau + b$, where d stands for distances, and τ stands for traversal times; a and b represent the slope and intercept respectively. Using this linear regression technique we obtained a *coefficient of determination* $R^2 = 0.83$ for Manhattan ($a = 3.63 \cdot 10^{-3}$, $b = -1.39 \cdot 10^{-2}$), and $R^2 = 0.85$ for Singapore ($a = 6.55 \cdot 10^{-3}$, $b = -3.40 \cdot 10^{-2}$).

APPENDIX C
FROM TIME CONDITIONS TO NECESSARY SPACE CONDITIONS

Claim 5.1 is central in our algorithm design as it allows us to re-formulate time-based queries in terms of (almost) equivalent, but more time efficiently computable, space-based queries.

Repeating the percentile preprocessing for different time values Δ_S (e.g. multiples of 5 minutes) we collected enough samples which are then interpolated to define γ-mappings for every continuous time value.

During our investigation we have also specialized the mapping γ for different districts within the city graph. This specialization turns out to be particular effective to capture the complexity and heterogeneous nature of speed networks, especially for maps from the old world (i.e. Europe, Asia), where there can be substantial differences in registered vehicle speeds in various city areas (i.e. downtown versus periphery). For details, see the Appendix to the original paper [1].

Leveraging the time-to-distance mapping γ and Claim 5.1, we can thus reformulate the time-based conditions for trip matches of SECTION IV into geometric-based conditions, as follows.

Let us start by considering the matches of the *first kind*, and refer to FIG. 2 (A) where the trips T_i and T_j are indicated with dashed lines and the solid line is used to denote the combined trip T_{ij}. Without loss of generality, we restrict ourselves to the case in which the origin of T_{ij} coincides with the origin o_i of the trip T_i.

The set of conditions $A(i,j)$ introduced in SECTION IV corresponds exactly to the (necessary and sufficient) conditions for the realization of a match of the first kind. In fact, two *necessary* conditions for $A(i,j)$ are:

$$\begin{cases} tt(o_i,o_j)+tt(o_j,d_i)\leq tt(o_i,d_i)+\Delta & (12)\\ tt(o_j,d_i)+tt(d_i,d_j)\leq tt(o_j,d_j)+\Delta & (13)\end{cases}$$

Indeed, Eq. 12 is obtained by subtracting the term st_i from both sides of Eq. 2, and remembering that $at_i = st_i + tt(o_i,d_i)$. Instead, Eq. 13 is derived from Eq. 3 by first subtracting st_i (as in the previous case) and by then considering that: $tt(o_j,d_i)+tt(d_i,d) < tt(o_i,o_j)+tt(o_j,d_i)+tt(d_i,d)$, as the traversal time $tt(o_i,o_j)$ is always a positive quantity.

In the symmetric case where the combined trip T_{ij} has its origin in o_j rather than in o_i, we can follow similar considerations to derive conditions $A(j,i)$.

Let then $D_i = \gamma(\Delta)$ be the the distance value associated to the tolerance parameter Δ of SECTION I, derived by means of the time-to-distance mapping γ. Notice that D_i depends on i whereas Δ does not depend on i; this is a direct consequence of the fact that the time-to-distance mapping γ has been specialized for different districts, and hence the distance D_i associated to Δ is not uniform within the city graph G_A. For simplicity of exposition, we have nonetheless removed any reference in γ to the district.

Furthermore, let $l_i = \gamma(tt(o_i,d_i))$ be the distance associated with the expected service time $tt(o_i,d_i)$ of trip T_i. The parameters D_j and l_j can be defined in a similar manner.

Because of the definition of γ-mapping, we can reformulate Eq. 12 and Eq. 13 in geometric queries specified in the following lemma. Here the reformulation is "approximate" because the γ-mapping reflects the time-to-distance relation up to the 95% of the paths in our city graphs. In the experimental section we will evaluate the accuracy of this approximation over our dataset of city graphs referring to Manhattan and Singapore.

Lemma C.1: It is possible to translate the feasibility condition $A(i,j)$ of a matching of the first kind into the following (approximately equivalent) geometric-based system of inequalities.

$$\begin{cases} d(o_i,o_j)+d(o_j,d_i)\leq l_i+D_i & (14)\\ d(o_j,d_i)+d(d_i,d_j)\leq l_j+D_j & (15)\end{cases}$$

We can provide a geometric interpretation of the first (resp. the second) inequality above: it corresponds to the membership of the node/point o_j (resp. d_i) to an *ellipse* which wraps entirely the trip T_i (resp. T_j) – see Figure 2-A.

We now focus on the matches of the second kind. Again, without loss of generality, we restrict ourselves to the analysis of the case in which the combined trip T_{ij} includes in its entirety the optimal (i.e. shortest) path for the trip T_j (the case of T_{ij} including the shortest-path for T_i boils down to analogous considerations). The feasibility of a combined match of the second kind is determined by the two conditions defined in $B(i,j)$ (see SECTION IV). Remembering that $at_i = st_i + tt(o_i,d_i)$, Eq. 5 for $B(i,j)$ can be re-formulated as:

$$tt(o_i,o_j)+tt(o_j,d_j)+tt(d_j,d_i)\leq tt(o_i,d_i)+\Delta \quad (16)$$

Now let a,b,c be three arbitrarily chosen nodes/locations in the city network. For the minimum traversal time $tt(a,c)$ from a to c, the relation $tt(a,c)\leq tt(a,b)+tt(b,c)$ must hold. Given this consideration as well as Eq. 16, we can derive the following system of inequalities:

$$\begin{cases} tt(o_i,o_j)+tt(o_j,d_i)\leq tt(o_i,d_i)+\Delta & (17)\\ tt(o_i,d_j)+tt(d_j,d_i)\leq tt(o_i,d_i)+\Delta & (18)\end{cases}$$

which represents a set of necessary conditions for the realization of Eq. 16, and hence necessary conditions for $B(i,j)$.

In the following, using again the notations D_i and l_i introduced above, we approximate the time-based conditions 17 and 18 via the geometric queries specified in the following lemma.

Lemma C.2: It is possible to translate the feasibility condition $B(i,j)$ of a matching of the second kind into the following (approximately equivalent) geometric-based system of inequalities.

$$\begin{cases} d(o_i,o_j)+d(o_j,d_i)\leq l_i+D_i & (19)\\ d(o_i,d_j)+d(d_j,d_i)\leq l_i+D_i & (20)\end{cases}$$

We can provide a geometric interpretation of this system of inequalities which correspond to the membership of the nodes/points o_j and d_j to an *ellipse* which wraps entirely trip T_i – see Figure 2-B. The ellipse for this second kind match is the very same ellipse that we discussed for first kind matching cases.

APPENDIX D
ALGORITHM DESCRIPTION

Below, we present the pseudocode of the two algorithms which implement our ride sharing solution. The pseudocode of ALGORITHM 1 logically precedes the pseudocode of ALGORITHM 2, i.e., the former must be executed before the latter.

In ALGORITHM 1 we use range searches in order to find – for each trip T_i – a list of destinations d_j falling in the geometric area defined by the ellipse of T_i (we will discuss more deeply the latter aspect in SUBSECTION D-C). In ALGORITHM 2 we run a similar procedure in order to find a list of origins o_j falling in the geometric area of trip T_i. The procedure of ALGORITHM 2 then computes for each trip T_i two *candidate sets* for a match of the first kind and of the second kind. These candidates are then verified by means of an explicit check of conditions $A(i, j)$ and $B(i, j)$. For the pairs (T_i, T_j) which result in a feasible match, the corresponding edge is inserted in the (initially empty) shareability network G_{SN}. The procedure of ALGORITHM 2 terminates with the execution of a maximum matching algorithm that computes the final set of combined trips.

Let us now dig into the algorithmic details of our approach by focusing in SECTION D-A on ALGORITHM 1, and in SECTION D-B on ALGORITHM 2.

A. Computing destinations close to each trip

We assume that trips in $\mathcal{T}$ are stored in a vector $\mathcal{T}[1, n]$ that is ordered for increasing st_* (starting time). Moreover, as already anticipated, we consider the *o*nline model in our study, and hence we fix the time window δ which defines, in a hypothetical ride sharing service, the maximum latency that a user can tolerate in waiting for an answer from the ride sharing system. Thus, for each trip T_i, we limit the search for trips T_j shareable with T_i to just those for which $|st_j - st_i| \leq \delta$.

For the concern – instead – of the oracle model (*aka* offline model, see SECTION III), we know that the time window parameter δ should be set to $+\infty$, or to a sufficiently high value. However, this might induce the unreasonable matching of trips which are far away in time. So it is better to limit our following discussion to the case of a finite δ, keeping in mind that the higher is its value, the larger is the computational cost of the algorithm for the creation of the shareability network, which would get increasingly denser. In conclusion, the actual value of δ in the Oracle model must be chosen carefully to trade computational time against the significance/optimality of the matching result.

In ALGORITHM 1 we maintain three indexes l, m and r ($l \leq m < r$) that iterate over the vector $\mathcal{T}$, and maintain the following invariant. At the m-th iteration (initially $l = m = 1, r = 2$), trip T_m is processed and the following conditions hold:

1) l is the *leftmost* index in the array $\mathcal{T}$ for which the relationship $st_l \geq st_m - \delta$ holds;
2) r is the *leftmost* index in the array $\mathcal{T}$ for which the relationship $st_r > st_m + \delta$ holds;
3) we denote by C_d the set of coordinates $p_* = (plat_*, plon_*)$, expressed in terms of latitude-longitude pairs, of destinations of the trips T_k for each $k \in [l, r-1]$.

Algorithm 1 Compute destinations close to each trip.

```
Require: 𝒯, δ, Δ
Ensure: DT, DT^{-1}
 1: n ← |𝒯|
 2: sort 𝒯 for increasing starting time
 3: DT and DT^{-1} are new inverted lists.
 4: C_d = ∅
 5: l = 1, r = 2
 6: for m = 1 to n do
 7:     while st_l < st_m − δ do          ▷ managing trips starting before trip T_m
 8:         ++l
 9:         remove d_l from C_d
10:     end while
11:     while r ≤ n and st_r ≤ st_l + δ do   ▷ managing trips starting after trip T_m
12:         insert d_r into C_d
13:         ++r
14:     end while
15:     R = compute_rectangle(m, Δ)        ▷ compute rectangle wrapping T_m
16:     N = range_search(C_d, R)          ▷ retrieve destinations close to T_m
17:     for each trip-identifier k with N do   ▷ insert in the inverted lists
18:         if scalar_product(d_m − o_m, d_k − o_k) > 0 then   ▷ cosine similarity filter
19:             insert k in DT[m]
20:             insert m in DT^{-1}[k]
21:         end if
22:     end for
23: end for
24: return DT, DT^{-1}
```

After each iteration:

- the index m is incremented by one, and $m \leq n$;
- the indexes l and r are (possibly) incremented to new values $l' > l$ and $r' > r$ which guarantee the conditions (1) and (2) above;
- the set C_d is updated by *removing* the coordinates of the destinations of the trips T_u, with $l \leq u < l'$; and by *inserting* the destinations of the trips T_v, with $r \leq v < r'$, so to maintain the condition (3) above.

During the m-th iteration, we perform a search for the *destinations* of trips T_k, $l \leq k < r$, which fall in the geometric proximity of trip T_m. We keep track of the retrieved points in the list $DT[m]$ described in SECTION V, which contains the identifier of trip T_a *iff* its destination d_a is "geometrically close" to trip T_m. While constructing DT, we also construct the inverted list DT^{-1} which stores in $DT^{-1}[m]$ the identifier of all trips that are "close" to the destination d_m of T_m. Therefore, $DT[m]$ is indexed by the full trip T_m, whereas $DT^{-1}[m]$ is indexed only by its destination d_m.

In Line 18 a further (geometric-based filtering) condition is

checked before the insertion of T_k into DT and DT^{-1}. More precisely, we discard trip T_k if its direction is rotated for more than 90 degrees with respect to the direction of trip T_m. This filter, which is both very simple and effective, is implemented by a simple check on the sign of the scalar product between the two vectors $d_m - o_m$ and $d_k - o_k$.

We have therefore proved that, for each pair of trips $T_k, T_m \in \mathcal{T}$, $k \in DT[m]$ iff $m \in DT^{-1}[k]$. It is worth remarking that trips are processed in an increasing order of identifiers, via the iterator m, hence the elements are inserted in DT^{-1} in increasing order, and so each list $DT^{-1}[m]$ is ordered too.

Line 15 makes use of a sub-procedure called `compute_rectangle` which takes as input the trip-identifier m, and returns a set of points describing the area "wrapping" trip T_m. Here, we are adopting a conservative approach which turns the search within ellipses into a search within rectangles, as illustrated in the previous Figure 2. This choice is dictated by the fact that the `Boost C++ Library` offers a template type, called `ring`, which allows to describe polygons (hence, rectangles) as a sequence of vertices. We then use `compute_rectangle` in Line 15 to return the rectangle "wrapping" the trip T_m. More details about this step will be provided in SUBSECTION D-C.

Finally, Line 16 makes use of procedure `range_search` that accepts as formal parameters: a set of locations C_d and a rectangle R (computed in the previous step via `compute_rectangle`) and returns the subset of locations in C_d which are contained within R. The set of locations retrieved by this call are the ones satisfying the first geometric relation of LEMMA 5.2 (or, equivalently, the first equation of LEMMA 5.3). In both cases, the role of trip T_m of ALGORITHM 1 is the one of trip T_i in the geometric relationship; the retrieved trip-identifiers correspond then to the trips T_j satisfying the two geometric relationships above.

B. Computing the shareability network

ALGORITHM 2 is executed after ALGORITHM 1 to populate the shareability network (SN) by leveraging the locality information stored in the posting lists DT and DT^{-1} (see Section VI for their definitions). This computation takes advantage also of the list OT, described in SECTION V and derived by following the same approach used for DT in ALGORITHM 1 as follows.

Specifically, we introduce the point set C_o containing the coordinates, expressed in terms of latitude-longitude pairs, of the origins of the trips T_k, with $k \in [l, r-1]$ (instead of trip destinations as in C_d). However, differently from DT, $OT[m]$ is computed *on the fly* at each iteration m, and discarded as m moves to the next value, so that the corresponding memory portion can be safely deallocated. Another difference between DT and OT is that the latter does not need its inverse OT^{-1}.

ALGORITHM 2 uses again the three iterators l, m and r, where the parameter m is the one driving the scan of DT and OT; and it maintains the same invariant as the one in ALGORITHM 1.

Hence, at iteration m, we have available OT, DT and DT^{-1}. These three inverted lists are passed to a function

Algorithm 2 Compute the shareability network G_{SN}

Require: $\mathcal{T}$, δ, Δ, DT, DT^{-1}
Ensure: G_{SN}
1: $n \leftarrow |\mathcal{T}|$
2: G_{SN} is a new undirected graph
3: $V_{SN} \leftarrow \{T_1, T_2, ..., T_n\}$
4: $E_{SN} \leftarrow \emptyset$
5: $C_o = \emptyset$
6: $l = 1, r = 2$
7: **for** $m = 1$ to n **do**
8: **while** $st_l < st_m - \delta$ **do** ▷ managing trips starting before trip T_m
9: $++l$
10: remove o_l from C_o
11: **end while**
12: **while** $r \leq n$ and $st_r \leq st_l + \delta$ **do** ▷ managing trips starting after trip T_m
13: insert o_r into C_o
14: $++r$
15: **end while**
16: $OT[m]$ is a new vector of trip-identifiers
17: $R = compute_rectangle(m, \Delta)$ ▷ compute rectangle wrapping T_m
18: $N = range_search(C_o, R)$ ▷ retrieve origins close to T_m
19: **for** each trip-identifier $k \in N$ **do** ▷ compute $OT[m]$ *on the fly*
20: insert k in $OT[m]$
21: **end for**
22: $C_{1k}, C_{2k} = find_candidates($ $OT[m], DT[m], DT^{-1}[m])$ ▷ Candidates for a match with T_m
23: $F_{1k} = check_candidates_1k(C_{1k}, m)$
24: $F_{2k} = check_candidates_2k(C_{2k}, m)$
25: **for** each trip-identifier $k \in F_{1k}$ **do**
26: insert (m, k) in E_{SN}
27: **end for**
28: **for** each trip-identifier $k \in F_{2k}$ **do**
29: insert (m, k) in E_{SN}
30: **end for**
31: **end for**
32: **return** G_{SN}

`find_candidates` which deploys geometric proximity information to determine the two sets of feasible candidates for a match with trip T_m of the first and of the second kind, as defined in SECTION V. ALGORITHM 2 uses the symbols C_{1k} and C_{2k} to denote these candidate sets. As a remark, we notice that a feasible match between T_m and T_k is discovered for both cases as follows: if the passenger of trip T_m is picked up as the first passenger (and hence before the one of T_k), then the candidate match is discovered at iteration m; otherwise, if the passenger of trip T_m is picked up as the second passenger after the one of T_k, then the candidate match is discovered at iteration k. This remark leads to the introduction of another invariant condition, which adds to the

three invariant conditions of ALGORITHM 1:

4) At iteration m we have already discovered all the trips T_k that could be combined with T_m and whose first passenger to be picked up is the one of T_k, with $k < m$.

The key property of our approach is that the set $C_{1k} \cup C_{2k}$ is argued to contain only a small fraction of the whole quadratic number of compatible trip pairs of the city graph under consideration. So the last step is to run a (costly) explicit check on the candidates in that set. This is implemented in procedures `check_candidates_1k` and `check_candidates_2k`.

The following subsections will explain the details of the procedures employed in ALGORITHMS 1 and 2.

C. Range queries

Range queries are at the core of ALGORITHM 1 (lines 15-16) and ALGORITHM 2 (lines 17-18). There, we need to compute first the parameters of the ellipse including the examined trip T_i, according to the rules of SECTION V, and then turn that ellipse into a rectangle because the `Boost C++ library` does not offer specific support for elliptic range searches. For the rectangle associated with the geometric-proximity region of a trip T_i, the following considerations hold:

- the point that bisects the diagonals of the rectangle corresponds to the center of the ellipse, and so to the midpoint of the segment that joins o_i and d_i;
- the rotation angle Θ of the rectangle and the one of the ellipse are the same; and
- the dimensions of the rectangle correspond to the dimensions of the axes of the ellipse.

FIGURE 2 pictorially represents some elliptic locality areas as well as the related rectangles. For the l_i and D_i values, we use the "typical distances" which are precomputed *off-line* for each location in the city graph using the methodology presented in SECTION V. We have indeed already seen that we leverage the time-to-distance mapping γ in order to: (a) dimension the ellipses (rectangles) wrapping each trip (SECTION V), and (b) execute range search queries within those ellipses/rectangles in order to find feasible candidates for a match (see the `range_search` procedure call in ALGORITHMS 1 and 2).

While solving the ride sharing problem, we often need to map *continuous* time values into the *discrete* set of precomputed distance values. In our implementation we proceed following this simple idea: we precompute typical *distances* for time values which are all multiple of 5 minutes, using the methodology of SECTION V. At query time, in the `compute_rectangle` procedure of ALGORITHMS 1 and 2 we first approximate each time values t to a the closest upper bound $t' \geq t$ which is multiple of 5 minutes; then, we estimate the distance d associated to t by using the typical distance d' precomputed for t'. Clearly this can be regarded as a conservative estimation, given that $t \leq t'$ implies $d \leq d'$, and hence we will query larger ellipses (rectangles), and retrieve consequently more retrieved candidates for a match.

Given the rectangle denoting the proximity region of T_i, the `range_search` procedure is implemented via an `R-tree` data structure, as the one offered by the `Boost C++ Library`, which is able to retrieve the points within the input rectangle in a time which is proportional to the number of retrieved points plus a term that scales with the logarithm of the number of indexed points (i.e. trips). `Boost` offers different ways to instantiate the `R-tree` data structure. One of the most important aspects of this setting is the choice of the algorithm for its load balancing: the more sophisticated algorithms result in a better balancing and hence in an improved query time, at the cost however of a worse update performance.

a) Combined trips of the first kind: Referring to FIGURE 2 (A), let us consider the trips T_R (colored *red*) and T_B (colored *blue*). As shown in the picture, and commented in SUBSECTION V-A, the combination is possible whether the destination d_R of T_R is contained in the blue ellipse and the origin o_B of T_B is contained in the red ellipse.

Using the inverted lists OT and DT, we can rephrase the previous two conditions for the combination of T_R and T_B in a trip whose origin coincides with the origin of the first trip, only if:

$$R \in DT[B] \text{ and } B \in OT[R] \tag{21}$$

or – equivalently – only if:

$$B \in DT^{-1}[R] \text{ and } B \in OT[R] \tag{22}$$

It is not difficult to notice that Eq. 22 corresponds to the geometric-based condition stated in LEMMA 5.2 of SECTION V.

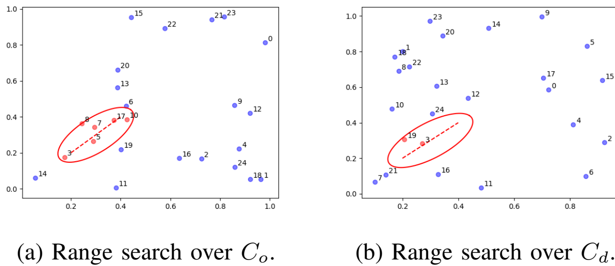
Hence, the retrieval of the candidates of the matches of the first kind boils down to computing the set intersection $DT^{-1}[R] \cap OT[R]$. This operation is very fast because we know that $DT^{-1}[R]$ is sorted, and $OT[R]$ can be sorted too, so that the intersection can be executed in a merge-sort-like fashion.

b) Combined trips of the second kind: Referring to FIGURE 2 (B), let us consider again the trips T_R (colored *red*) and T_B (colored *blue*). In order to generate a combined trip of the second kind, the destination d_B of T_B and the origin o_B of T_B must be both contained in the red ellipse. By leveraging the inverted lists OT and DT, we can rephrase these conditions as:

$$B \in DT[R] \text{ and } B \in OT[R] \tag{23}$$

This equation corresponds exactly to the geometric-based condition stated in LEMMA 5.3 of SECTION V.

Hence, the retrieval of the candidates for the second-kind match boils down to computing the set intersection $DT[R] \cap OT[R]$. This operation corresponds to the retrieval of all trips which have both origin and destination falling within the ellipse surrounding trip T_R. In FIGURE 8 it is depicted an example in which we represented with red dots the (A) origins and (B) destinations retrieved during the range searches within the ellipse defined by trip T_R (which is represented by the red dashed line).



(a) Range search over C_o. (b) Range search over C_d.

Fig. 8: (a) and (b) depict a range search over the sets C_o and C_d, respectively. Every dot in (a) corresponds to the origin of a trip $T[k]$ with $k \in [l, r-1]$, whilst every dot in (b) represents a valid destination for those T_k. Every dot is annotated with the *id* of the corresponding trip. The valid candidates for $B(i, j)$ are obtained by intersecting trip-ids retrieved by the two range queries. According to Figures (a) and (b), only trip T_3 satisfies this property, which corresponds to condition 23.

At the m-th iteration of ALGORITHM 1, the examined trip is T_m and the range search of FIGURE 8 (B) is executed over the set C_d which includes the destination of all trips T_k with $k \in [l, r-1]$. This computation creates the set $DT[m]$.

At the m-th iteration of ALGORITHM 2, the examined trip is T_m and the range search of FIGURE 8 (A) is executed over the set C_o which includes the origin of all trips T_k with $k \in [l, r-1]$ (see definition in SUBSECTION D-B). This computation creates the set $OT[m]$ (line 16).

These two sets of trip identifiers are then used in the final part of the procedure ALGORITHM 1 to retrieve the candidates for a combined match of the second kind.

One more time, the set intersection (between $OT[m]$ and $DT[m]$) can be executed in a merge-sort-like fashion. For this, we need first to sort $OT[m]$ and $DT[m]$. We have that $OT[m]$ is already sorted (we have sorted it while computing first kind candidates), and $DT[m]$ can be sorted too.

As a final remark, we point out that subsections a) and b) have explained how the two sets $DT^{-1}[m] \cap OT[m]$ and $DT[m] \cap OT[m]$ $\forall m = 1, 2, ..., |\mathcal{T}|$, can be computed so that the final part of ALGORITHM 2 can eventually retrieve the candidate set of all first and second kind matches.

D. Further (on-the-fly) matching

Our algorithm can also be easily and seamlessly extended in order to include on-the-fly matching with already matched trips. Indeed, even when considering matched user, it is actually possible for that user to sit in a empty-cruise taxi for a portion of his travel towards the destination. Thus one can think to extend matching conditions, which are expressed in terms of Δ in SECTION IV, in order to try to operate further matching for those (empty) taxi cabs. It is enough to add in the next batch of taxi requests also some mock requests, which describe those partially combined paths, which can be further combined. For these portions of paths:

- we already know the origin and the destination, as well as the time schedule, i.e. starting and arrival times
- we re-compute the maximum tolerated delay as $\Delta' \leq \Delta$, taking into account that the taxi user is already facing some delay on the previously matched trip.

This dynamic approach offers the advantage to increase the vehicle utilization, thus possibly leading to further fuel and money savings. On the other hand, it is worth to notice that dynamic approaches increase also the user discomfort which have to face larger path delays, and also see their scheduled path possibly re-scheduled abruptly while they are experiencing the service. For these last reason we have not further investigated the on-the-fly matching in the remainder of this paper, hence limiting each taxi user to *at most* one single match with any other user.

APPENDIX E
TIME AND SPACE COST

In the following two subsections we dissect the asymptotic time and space complexity of our proposal in their main parts. We will also compare this asymptotic cost with the one required by the legacy all-to-all shortest path algorithm adopted in [1].

A. Building time

First of all, we notice that our algorithm deploys some sophisticate data structures, such as the R-tree or the Contraction Hierarchies (CH). The former is a well-known two-dimensional range queries data structure. The latter are pre-computed *una tantum* and then used by ALGORITHMS 1 and 2.

For evaluating the cost of the building time, we need to take into account the following two contributions:

- The first one is given by the construction of CH, which is a speed-up technique for finding shortest-paths within a road map. A central component of the preprocessing consists of computing a so-called contraction order by means of *shortcuts*. Shortcuts enable to exploit some pre-computed distances between important junctions within the city graph so that, at query time, there is no need to explore the full path between these junctions thus saving time in its distance calculation. Given a city graph G_A and given an integer K, the problem of finding a node ordering for G_A such that the resulting contraction hierarchy for G_A has at most K shortcuts is an NP-complete problem [23], and hence it is approached through heuristics. The precomputation for CH turns out to be very fast, as it takes few hundreds of milliseconds for both the city graph of Manhattan and Singapore.
- The second one is the time needed to precompute the time-to-distance association γ. This can be easily performed by means of a pruned Dijkstra search around each of the crossroads in the input city graph. Since these statistics are used within our geometric-based filter, their computation could tolerate some inaccuracies, so it is possible to speed up the γ computation by means of some proper sampling of the crossroads where these statistics are estimated. For instance, it is possible to sample a sufficiently large amount of random paths to derive a linear approximation for γ, as it is reported

in FIGURE 7. Furthermore, there is no need to store percentile values for each of the possible crossroads in the city graph. In fact, one could consider clustering groups of neighboring crossroads/nodes, and then select for each of these clusters a representative node which assumes the role of *centroid* for the cluster. Instead of keeping statistics for all nodes, it is then possible to restrict the time-to-distance association analysis to just those centroids, thus saving both precomputing time and occupied space.

B. Time complexity

Once the data structures have been built, ALGORITHMS 1 and 2 start their computation, and their overall time complexity can be estimated as follows.

- For what concerns the shortest-path computations we leverage the *Contraction Hierarchies* (CH) technique [21], whose API has been recently enriched in order to optimally solve many-to-many shortest-path calculations, as needed by the ride sharing problem. It turns out that both the legacy and the new approach execute $\Theta(|\mathcal{T}|)$ total CH queries. However, the number of involved end-points in the two cases is remarkably different, as in the legacy algorithm all the path-computations are 1-to-$|\mathcal{T}|$ (or $|\mathcal{T}|$-to-1), whereas our new approach restricts the computation to a narrower set of C candidates, and hence we run 1-to-C (or C-to-1), where C is not greater than $|\mathcal{T}|$, and possibly much smaller than $|\mathcal{T}|$ ($C << N$).
  It goes without saying that the literature offers other solutions to many-to-many shortest path calculations, which are alternative to CH, and could be used for path-queries. These other solutions could be adopted in place of CH without changing the overall algorithmic structure of our proposal, in which that computation is considered as a black-box.
- The R-tree is a very efficient data structure for the retrieval of points/objects within a rectangular area in the Cartesian plane. Thanks to its self-balanced structure, the R-tree guarantees to serve a range search in $O(K \log U)$ time, where K is the number of retrieved trips and U corresponds to the set of points on which the range-search is performed. In our case, U is either $|C_o|$ or $|C_d|$, and they are trivially upper bounded by the number of input trips $|\mathcal{T}|$. This upper bound turns out to be a good estimate in the case of the *Oracle model*, in which we are going to consider many more trips than in the *Online model*, including those ones that will appear in a possibly far "future". The number K has been denoted with $|OT[i]|$ and $|DT[i]|$ in the previous sections. Consequently, the asymptotic cost for every range query on the R-tree is $O((|OT[i]| + |DT[i]|) \log |\mathcal{T}|)$.
- Computing the set of feasible trip pairs requires the evaluation of Conditions (22) and (23). This needs to execute the intersection between lists $OT[i]$ and $DT[i]$. We have already mentioned that a fast way to implement this intersection is to first order those lists and then proceed by means of a merge-based procedure which takes linear time in the cardinality of the two merged sets. Overall, the cost of an intersection is dominated by the cost of the two sorting steps, thus $O(|OT[i]| \log |OT[i]|)$ and $O(|DT[i]| \log |DT[i]|)$. This cost is in turn dominated by the cost of the range-search queries above, and therefore these sorting operations have no impact in the asymptotic time analysis of our algorithm.
- Finally, we consider the cost of executing the *maximum cardinality matching* (MCM) algorithm over the shareability network G_{SN}. The time complexity of the Edmond's MCM algorithm is asymptotically equal to $\Theta(m\, n\, \alpha(m, n))$, where m and n are the number of nodes and edges (respectively) inserted into G_{SN} by ALGORITHM 2, whilst $\alpha(\cdot)$ is a slow growing function that is at most 4 for any feasible input size. As already reported in the appendix to [1], the MCM algorithm takes in practice just few seconds on the city of Manhattan and thus can be considered negligible.

Let us denote with T_{MCM} the time complexity for the maximum matching calculation step. As just seen, the actual cost T_{MCM} depends on the dimension (in terms of number of nodes and edges) of the SN.

We also denote with T_{CH} the *worst case* time complexity for a one-to-many distance computation within the city graph.

We finally have that the time complexity for the *legacy* solution enriched with the CH strategy for path-queries can be characterized by:

$$T_{old} \leq N \cdot T_{CH} + T_{MCM} \tag{24}$$

whilst our novel solution solves the same problem in:

$$T_{new} \leq N \cdot T_{CH} + T_{MCM} + N\, C\, \log N \tag{25}$$

Let's comment out the result of Eq. 24 and 25 above. In the equations we have used $N = \mathcal{T}$ to denote the number of trip requests; C denotes instead the average number of (first and second kind) candidates retrieved at each iteration of ALGORITHM 2 ($C << N$). The logarithmic term in T_{new} is introduced by the $\Theta(N)$ r-tree queries executed in ALGORITHMS 1 AND 2. The logarithmic term depends on the number on the input dimension, and can be *de facto* upper bounded by a small constant (e.g. 13 for a 20 minutes batch of requests).

Comparing the formulas of Eq. 24 and 25, we can state that the novel algorithm is *not slower* than the legacy one. Actually, it is reasonable to expect (we will see it in the experimental section) that our algorithm performs better than the legacy one, for a few different reasons: (a) we have hidden in the common worst case T_{CH} notation the fact that one-to-many queries have different cardinality in the two approaches (1-to-N in the legacy, 1-to-C in the novel, with $C << N$), (b) the computed paths are usually much shorter (and hence faster to compute) in the novel approach, as are bounded by locality areas, and (c) the MCM algorithms runs faster in the novel approach as it is executed over a pruned G_{SN}.

C. Space complexity

As far as the space occupancy is concerned, we observe the following:

- For what concerns the *Contraction Hierarchies* (CH) technique, the authors of [21] demonstrated that their solution offers a good compromise between speed and amount of data generated during the preprocessing phase. In particular, they have shown that in many practical situations, CH occupies *less* space than the input graph itself. Thus, asymptotically speaking this space cost is *de facto* dominated by the space needed to store the travel estimates over the road segments.
- At *query time* the search of CH can indeed be regarded as a bidirectional version of Dijkstra's algorithm, operating an upward and a downward search within two proper parts of the input graph and of the precomputed hierarchies. In particular the authors of [21] demonstrated that CH visits only a few hundreds vertices (out of ten millions) even when running on top of huge continental road networks [21]. One should also notice that the auxiliary information computed for the city graph G_A are exactly the same under the legacy and the new approach. Again, both the legacy and the new approach need to execute $\Theta(|\mathcal{T}|)$ CH computations, but with different cardinality: 1-to-$|\mathcal{T}|$ in the case of the legacy approach, and 1-to-C in the case of our new approach (with $C << |\mathcal{T}|$, see above).
- The R-tree is built on the trip set $\mathcal{T}$ and thus it takes $O(|\mathcal{T}|)$ space.
- Another contribution to space occupancy is the one related to the construction of the postings list OT and DT. As described in SECTION VI, we initially build a full posting list for all the destinations close to each trip and then, in a second phase, we determine on-the-fly the origins close to each trip. If we use the symbols $|DT|$ and $|OT|$ to denote the (average) length of each of these posting lists, we can conclude that the space occupancy of our solution includes also a term $O(|\mathcal{T}| \cdot (|DT| + |OT|))$. Since $|DT|$ and $|OT|$ are in practice few hundreds, then this additional space cost is $O(|\mathcal{T}|)$.
- The previous linear term applies also to all other data structures that our filtering algorithm builds and processes. In fact, we are referring to: (1) the lists of candidates for a match of the first and of the second kind, which are determined via an intersection among the postings lists OT and DT, and (2) the final pairs of trips which are confirmed to be valid matches either of the first or of the second kind, and which are clearly a subset of (1). Since those final matches correspond to the edges of the *shareability network* G_{SN}, we can also state that G_{SN} occupies $O(|\mathcal{T}|)$ space.
- Other space is occupied by the auxiliary data structure that stores the statistics (95-percentiles) about the typical Euclidean distances of shortest-paths outgoing from each vertex in the road network, and whose duration can be bounded within multiples of Δ minutes. We store those distance values for just few multiples of Δ: in our code, we have considered just $5, 10, 15, ..., 55$ and 60 (namely, 12 different samples). Therefore, we can state that the space occupancy for these statistics is just $(12 \cdot 4) \cdot N$ bytes, which leads to the asymptotic notation $O(N)$, where N is the number of nodes in the city graph.

Overall we can conclude that, let S_{CH} and S_{MCM} denote the space complexity for the one-to-many CH path computations over G_A, and for the maximum matching execution over G_{SN}, the asymptotic space occupancy of both the legacy and our new approach can be bounded by $S_{CH} + S_{MCM} + \Theta(|G_A| + |\mathcal{T}|)$. So our solution is not worse than the legacy one by [1].

Digging further into the algorithmic details of the two algorithms, it can be shown that the described worst-case scenario is very pessimistic in practice as, indeed, our solution is more succinct in space than the legacy solution. In fact, we have hidden in the notation S_{CH} the dependency on the number of selected destinations in the 1-to-many shortest path computations. As this number increases, CH visits more nodes in the city graph and, possibly, these nodes are spread over a large geographical area. Conversely, our approach selects just $C << |\mathcal{T}|$ candidate trips (hence destinations or sources), which are very short because they are selected only among the ones clustered within a (narrow) locality area wrapping T.