RETRy: IntegRating RidEsharing with Existing Trip PlanneRs

Ali Masri
VEDECOM
77 Rue des Chantiers
Versailles 78000
ali.masri@vedecom.fr

Karine Zeitouni
University of Versailles
Saint-Quentin-En-Yvelines
45 Avenue des Etats-Unis
Versailles 78000
karine.zeitouni@uvsq.fr

Zoubida Kedad
University of Versailles
Saint-Quentin-En-Yvelines
45 Avenue des Etats-Unis
Versailles 78000
zoubida.kedad@uvsq.fr

ABSTRACT

Ridesharing services are getting a lot of attention in the recent years as they are beneficial for both travelers and drivers, and friendly to the environment. The problem is that these services are isolated from existing public transportation networks. They are proposed as alternative plans and not as part of a trip plan. Integrating these services may vastly improve the trips quality and serve as a backup plan in case delays or unexpected events happen. The main challenge facing the integration is limiting the search space and coping with the specific characteristics of ridesharing. This paper introduces RETRy, a system that enables the integration of ridesharing services with existing trip planners to provide real multi-modal trip planning solutions in near-real time.

CCS CONCEPTS

•Applied computing → Transportation;

KEYWORDS

Multimodality, Trip Planners, Ridesharing

ACM Reference format:
Ali Masri, Karine Zeitouni, and Zoubida Kedad. 2017. RETRy: IntegRating RidEsharing with Existing Trip PlanneRs. In *Proceedings of SIGSPATIAL Conference, California, USA, November 2017 (SIGSPATIAL'17),* 10 pages.
DOI: 10.1145/3139958.3140022

1 INTRODUCTION

Trip planning applications are becoming one of the essential applications we regularly use in our daily lives. They help us plan better trips that are faster, cheaper and more secure. Many research works were conducted over the years to provide better trip planning algorithms. Up to this point, the algorithms gave good results on the data especially when they are bounded to public transportation networks. However, with the need of integrating new transportation data and services, existing algorithms became insufficient to support the new requirements. Many new services are proving to be very efficient for travelers. Ridesharing is an example of these services and the scope of this work.

Ridesharing is a transportation service where individual travelers share a vehicle for a trip and split travel costs with other travelers who have similar itineraries and time schedule. These services are getting a lot of attention in the recent years as they are beneficial for both travelers and drivers, and friendly to the environment [11].

Ridesharing can be a solution for areas not covered by public transport or a backup plan in case of some perturbations. It has many advantages that benefit travelers, drivers and the environment. Some of ridesharing advantages include cost saving, lower parking demand, lower emissions, better urban design, less congestion and less vehicle travel.

The main problem is that these services are not yet fully integrated in existing trip planning algorithms. They are proposed as alternative plans if no public transportation plan is found or simply not preferred. For example, consider a traveler who plans to travel from a source to a destination. Existing solutions may suggest the following plans. The first is a sequence of multimodal trips e.g. walk from source to bus stop 1 → take bus from bus stop 1 to a point x → walk from x to train stop 1 → take train from train stop 1 to point y → walk from y to bus stop 2 → take bus from bus stop 2 to point z → walk from z to destination. The second is a car sharing solution that simply suggests: walk from source to pickup location → take the car sharing from the pickup location to the drop-off location → walk from the drop-off location to destination. We notice here that the car sharing solution is proposed in isolation of the multimodal solution and not really as part of the trip.

Integrating these services within the existing trip planning solutions may vastly improve the quality of the trips. In addition it may serve as a backup plan in case of some delays or unexpected events - which may happen in public transportation services.

The main cause behind the isolation is that it is simpler to calculate trips with scheduled networks due to the fact that we know in advance the departure and arrival times of each transportation unit. This is not the case in ridesharing services where a driver may at any time notify an intention of sharing a ride with others. This complicates the problem because these requests will appear/disappear dynamically on the network. In addition, the ridesharing problem is complex on its own [11, 22] and adding the connection with other modes will vastly increase the size of the search space, and will result in more complex calculations.

In this work, we seek for a way that integrates the complex characteristics of ridesharing services with existing trip planning algorithms. Our goal is to provide an algorithm that is able to handle efficiently the dynamic nature of these services. Many challenges arise from this goal which we summarize as follows: 1) adding the new services will enormously enlarge the search space, 2) there is no fixed pickup and drop-off stops for ridesharing services as in public transportation services, 3) there is no information about the future position of the ridesharing cars, 4) the additional cost added by integrating the new services should be taken into account and finally 5) there is no clear way on where we should integrate the services on the trip timeline.

This paper introduces RETRy, an approach enabling the integration of ridesharing transportation services with existing multimodal public transportation networks while reducing the search space. The idea in general is to calculate a trip using existing trip planners and use it as a baseline. We then use the fixed origin, destination and the public transportation stops in this *reference trip* as possible pickup and drop-off parameters for ridesharing services, trying to optimize the trip by introducing more optimal sub-trips by using this mode. The resulting trip forms the initial plan for a passenger to follow. During the trip execution the trip can be re-planned as a response to unplanned events, which makes it adaptive and reliable. The feasibility of the trip is calculated by the time both drivers and passengers may wait for the pickup. Two approaches are proposed for reference trip generation and are discussed later in the paper. The first uses only the base plan generated by trip planners, while the second uses the K-nearest trips as base plans. We have evaluated the approach by issuing random trips with different modes and comparing the gain we have got in each mode. The evaluated modes are: public transportation only, public transportation with ridesharing and ridesharing only. The comparison between the different modes shows that integrating ridesharing with public transportation networks results in faster trips with little additional cost especially with the K-nearest trips approach.

The paper is structured as follows: in Section 2 we list the recent approaches in trip planning and ridesharing. Section 3 introduces the RETRy framework and details its components. Section 4 presents the implemented system and an evaluation on a real use-case. Finally, a conclusion and discussion of future work are provided in Section 5.

2 STATE OF ART

Many trip planning algorithms were developed along the years. They can be decomposed based on the way they approach the problem and the solutions they provide. Some algorithms only target the problem as a routing problem on road networks. These algorithms are suitable for walking or driving passengers who seek a guide on how to traverse their routes efficiently. Other approaches target trips including public transportation services such as trains, buses, etc. They use timetable and infrastructure data to plan optimized trips for passengers. Passengers may use these trips to reach their destinations using a combination of public transportation services. On the other hand, with the appearance of on demand transportation services such as ridesharing, new algorithms were developed. The goal of these algorithms is to match service requests with service offers.

In this section we introduce the latest approaches in the fields of route planning, timetable trip planning, and ridesharing matching algorithms.

2.1 Route Planners

Route planning is an extensively studied field with a lot of contributions that are aimed at building fast and optimized algorithms for navigating routes. The shortest path problem is one of the most famous problems in this field. It is very popular especially in trip planning since basically the shortest trip is what people ask for. In the following we will present some of the most used approaches for solving the shortest path problem.

Dijkstra's algorithm [9] is a well-known shortest path algorithm. It works by maintaining a priority queue of vertices ordered by distance from a source point. The algorithms begins with initializing all distances to infinity except the distance to the source which is initialized to zero and added to the queue. After the initializing phase the algorithm stars the iteration phase. On each iteration the algorithm extracts from the queue the vertex with the minimum distance and scans it. It looks at all of its neighbors and determines the distance to them. If the value improves the distance it is then updated and the edge is relaxed. Dijkstra is a label setting algorithm meaning that labels are only scanned once and the algorithm terminates once reaching the target.

Bellman-Ford [2] is another well-known shortest path algorithm. It does not use priority queues as in Dijkstra. Instead, it works in rounds by scanning all vertices whose distance labels have improved. The vertices to be scanned are stored in a simple FIFO queue. Bellman-Ford is a label correcting algorithm meaning that labels can be scanned more than once. It is often much faster than Dijkstra and has an advantage of working on graphs with negative edge weights.

The next algorithm in the list is Floyed-Warshall algorithm [10]. It works by computing the distances between all pairs of vertices. Eventhough it may sound expensive, it is efficient for use on dense graphs.

A* algorithm [12] is a well-known algorithm that uses goal oriented techniques. In short, these techniques aim to target the search towards the goal to avoid the scan of unnecessary vertices. It uses a potential function on the vertices then runs a modified version of Dijkstra. In addition, it can be run bidirectionally to speed up the process.

Many other contributions were presented in the field of route planning. They are classified based on: i) basic techniques that work by scanning all vertices as in the top three mentioned algorithms, ii) goal oriented techniques as seen in A*, iii) separator techniques that uses separators to decompose the graphs into sub graphs, iv) hierarchical techniques that exploits the hierarchy of road networks, v) Bounded-Hop techniques that use pre-computation to add virtual shortcuts to the graph speeding up the search process and finally vi) approaches that use combinations of the above techniques. Readers are advised to look at [1] for deeper insights on the mentioned techniques and approaches.

2.2 Timetable-based Planners

Timetable planners aim to find trips from a source to a destination using a combination of nearby services using transit stops. In general, the timetable information require remodeling the network graph to be able to represent timetable information. Two main approaches have been proposed for modeling the shortest path problems in timetable systems: the time expanded approach and the time dependent approach.

In the time expanded approach, multiple nodes are constructed that correspond to a specific time event (departure or arrival) at a station while the edges are the connections between two events in the network. The result is a very large graph that contains all possible connections at different times according to the timetable. The authors in [21] adopted the time expanded approach to optimally solve the shortest path problem on a static graph. Furthermore, The time expanded approach was extended in to solve multi-criteria problems in [17].

The time dependent approach avoids the existence of multiple nodes per station. It models the network as one node per station with multiple edges representing the possible different times for events between two stations. The authors in [6] were the first to target the shortest path problem on time-dependent graphs. Later, the approach was generalized to support multi-criteria by the authors in [14]. An important study was presented in [18, 19] to investigate the complexity of the shortest path problems and give the required algorithms.

The connection scan algorithm (CSA) [8] is a shortest path algorithm that uses the time expanded model. It works by receiving a stream of connections ordered by departure time and chooses the fastest way to reach one stop from another. Due to the fact that the connections are pre-sorted and can be accessed one by one in a single iteration, CSA is faster and more scalable than other existing algorithms. RAPTOR [7] is an another interesting algorithm that optimizes the number of transfers in the Pareto-sense in addition to the arrival time. SUBITO [3] is an accelerated version of Dijkstra applied to the time dependent graph model. Instead of scanning all the nodes, SUBITO uses lower bounds on the travel time to prune the search space. The authors in [23] proposed a preprocessing-based algorithm to this problem. It computes all possible transfers between trains in order to speed up the query time. However, the preprocessing time is large which leads to an extension of the approach in [24] to achieve higher speedups.

2.3 Ridesharing Matching Algorithms

A ridesharing request consists of two points and two constraints. The points specify the pick-up and drop-off positions and the constraints specify the waiting time and a service constraint. The waiting time constraint is the maximal time the rider can wait after making the request. The service constraint is the acceptable extra detour time from the shortest possible trip duration. The main problem in ridesharing is: given a set of cars on the road network, we need to match a rider's request to a car that can satisfy all the constraints we previously mentioned.

The main challenge in dynamic sharing systems is the ability to handle large number of trip request and cars in real-time. This is due to the dynamic movement of cars and riders and the requirements of handling requests in matter of seconds. The Filter and Refine framework [22] facilitates the problem by splitting it into two main problems and allowing us to conquer the problem on a smaller scale. In the Filter phase, the framework filters all the drivers that do not match the request criteria. While in the Refine phase, an appropriate algorithm is applied to get the matching pairs taking into account the constraints.

An example of a filtering approach can be viewed in [15], where the authors use a grid-based index to partition the map. A grid distance matrix is constructed to fill-out all the distances between the grids. Each grid index contains three main sets: a spatial set indicating a the set of grids from nearest to furthest from a geospatial point of view, a temporal set indicating the set of grids from the nearest to the furthest from a temporal point of view and finally a set of vehicles that will enter the grid in the future with a time window of two hours. These indexes are then used to quickly filter-out cars that can not be matched to requests. The limitation of this approach is that the pre-computations are costly are not suitable for large scale ridesharing.

In SHAREK [4], the authors proposed three main pruning techniques to minimize the need for shortest path calculations. The used pruning techniques are: Euclidean temporal pruning that performs a range query to filter far cars, Euclidean cost pruning that filters-out cars that do not match the given cost constraints and finally a Semi-Euclidean skyline-aware pruning that selects candidates by balancing the cost vs time constraints.

Some approaches use techniques such as Branch and Bound [20] or Integer Programming [5], however, the problem with these approaches is the fact that each request requires the rescheduling of all previously computed schedules. Therefore, there is no use of previous computations.

The authors in [13] propose a tree structure to preserve previous computations and handle new requests as an insertion to the tree. The resulting solution may not be the optimal. However, it is very fast to compute. This reduces the complexity of the matching algorithms from O(k!) to O(k^2) where k is the number of requests. The limitation of this approach is that once the tree is computed, it is not updated in real time which makes the computation to be outdated.

2.4 Discussion

The domain of trip routing is rich with many approaches. Route planners [1, 2, 9, 10, 12] provide good solutions for solving the shortest path problem. However, they are not enough to represent and plan trips in dynamic graphs that contains timetable information. Timetable planners [3, 7, 8, 23] solve the timetable problem by introducing time-dependent graphs and the required algorithms to optimally traverse and plan trips based on them. These algorithms are used in various trip planning applications. New approaches that target on-demand transportation services such as ridesharing [4, 13, 15, 22], focus on the problem of matching riders with passengers. Therefore, the proposed trips are isolated from the multimodal trips and offered as alternative solutions. Our ultimate goal is to be able to integrate ridesharing within current trip planning approaches to enable real multimodality. Many challenges arise from this goal where the matching becomes very complex since both the user and the car are moving. In addition, there is always uncertainty of the departure and arrival times, unexpected cancellations and delays in addition to the complexity of the algorithm regarding the large network graph. Our ultimate goal is to overcome these limitations and provide a solution with real multimodal planning that integrates the new services with existing public transportation ones.

3 INTEGRATING RIDESHARING INTO TRIP PLANNING

Integrating new transportation services with current trip planning solutions will add many challenges and performance issues to the standard trip planning problem. The search graph will be a huge and dynamic one with services appearing/disappearing at random. There will be no clear position to issue a service from especially when the stops are not fixed. In order to solve this problem we introduce RETRy. RETRy is a framework enabling the integration of ridesharing and public transportation services. The general idea is to use existing public transportation trip planning algorithms [1] to generate a base plan that we call a reference trip. Then we try to improve the reference trip by injecting ridesharing services when possible. By doing so, the search space is reduced a lot and the positions of the requests is bounded to the transit transportation stops. In addition, RETRy makes it possible to adapt to changes by a repeated execution of the process over the course of the trip.

The RETRy framework is composed of four main components: (i) a query interface for users to enter their trip information, (ii) a multimodal planner to help in calculating reference trips, (iii) a ridesharing service to get the offers and (iv) the core planner.

This process is not straightforward for many reasons. First of all, reference trips do not always exist. Furthermore, the future positions of the ridesharing cars is unknown in advance since a ridesharing service is dynamic. Finally, optimizing the selection of pick-up and drop-off positions for the ridesharing request is a combinatory problem which is complex in its nature.

To cope with these problems, our proposal uses a heuristic approach. In the following, we present the approach in details. The first section discusses the first part of the algorithm, which is generating the reference trip. The second section targets the optimization of the generated reference trip with the injection of ridesharing services.

3.1 Generating Reference Trips

Reference trips can be calculated by existing trip planning algorithms [1] that are already optimized for public transportation routing problems. A reference trip is a set of sub-trips, where each sub-trip is assigned to different transportation units. A sub-trip is composed of: departure stop, departure time, arrival stop and arrival time. Given a user's location and destination, our planner asks the existing trip planner to generate a reference trip combining a set of sub-trips to match the user's query.

However, it is not always possible to find a reference trip simply due to lack of existence of nearby services. To cope with the problem, we provide an alternative method. We propose to calculate a sub-reference trip, which is a trip enabling the traveler to reach the closest point of his destination from the closest point of his origin. A sub-reference trip is intended to utilize public transportation as much as possible, thus reducing the walking time for passengers.

To generate a sub-reference trip we propose two different approaches. In the first one, the planner performs two range queries. The first query selects all source's nearby public transportation stops as starting points. The second range query selects the destination's nearby public transportation stops to be considered as destination points. After having both source and destination points, we issue trip requests from the combination of both points until a trip is found. The second approach is to choose the midpoint, and issue two queries from the starting point to the midpoint then from the midpoint to the destination. This process is done recursively until a solution is found. Our approach is inspired by the idea of selecting nearby grids in [15] and the time-dependent range query introduced in [4].

We note that, at this step, these trips are not optimized and may contain a lot of walking gaps that are infeasible for the traveler. for this reason, we introduce the next step of the core planner which tries to optimize this reference trip using the ridesharing component.

Algorithm 1 shows the trip generation algorithm according to the midpoint method described above. The algorithm starts by calculating a reference trip (line 1). The caclulateTrip function calls a public transportation trip planner with the source and destination points in addition to the departure time. If the reference trip was successfully found, the algorithm terminates by returning the calculated trip. Otherwise, the algorithm gets the midpoint between the source and destination (line 5). Two more trips are issued on lines 6 and 7 that represent a trip from the source to the midpoint (left trip) starting from the departure time and a trip from the midpoint to the destination (right trip) starting from the left trip's arrival time. It is important to note the recursive calls for these trips. This is due to the possibility of also not finding trips for these two parts, thus the algorithm works recursively by computing the midpoint in each part and calculating left and right trips. Finally, the algorithm combines both left and right trips and return the results (lines 8 and 9).

```
   Data: source: src, destination: dst, time: departure time
   Result: a trips from a source to a destination
   /* Calculate a trip from source to destination
      using existing multimodal planner                  */
1  $trip \leftarrow calculateTrip(src, dst, time)$;
2  if $trip$ not empty then
3  |   return trip;
4  end
   /* If no trips were found, get the midpoint of the
      trip and issue the queries from source to
      midpoint and from midpoint to destination
      recursively                                        */
5  $midpoint \leftarrow getMidpoint(src, destination)$;
6  $lefttrip \leftarrow calculateTripRecursive(src, midpoint, time)$;
7  $righttrip \leftarrow$
     $calculateTripRecursive(midpoint, dst, lefttrip.arrivaltime)$;

   /* Combine both sub-trips to get the overall one */
8  $overalltrip \leftarrow combinetrips(lefttrip, righttrip)$;
9  return $overalltrip$;
```

Algorithm 1: Trip Generation Algorithm

3.2 Multiple Reference Trips

The reference trips generated by public transportation trip planners are highly dependent on the state and the infrastructure of transportation networks. The reference trip only takes into account public transportation systems neglecting the fact that alternative trips may perform better when integrated with other modes.

For this reason, an alternative method is proposed to take advantage of this possibility. Instead of querying just the nearby stop, we choose the K-nearest neighboring stops from the source and issue K queries to the destination. Even-though some stops may be far away from the starting point, there is a possibility that by reaching the stop faster with ridesharing, the resulting overall trip will be faster to the destination point. This modification will result in multiple reference trips that will be taken by the next step of the RETRy framework in order to be optimized with ridesharing services. In addition, instead of querying directly to the destination point, we may use the same technique and query the K-nearest stops and issue queries to them. Finally, we add the walking path from the source to the initial stops and from the final stops to the destination. The resulting trips are then passed to the optimization phase.

Algorithm 2 shows the extended approach. It first starts by initializing an empty trips array (line 1). Later, the K-nearest stops to the source point are queried (line 2). The algorithm loops on the list of stops (line 3) and calls algorithm 1 to generate a reference trip from the each nearby stop to the destinations (line 5). Note that the departure time is the initial departure time given by the passenger in addition to the walking time to reach the stop. Thus, we calculate the walking time from the source to the stop and query based on the new time (line 4). All the generated trips are appended to the final trips array. Before the algorithm terminates, it adds a walking path from the source point to each nearby stop (line 6). This path will be later optimized by a ridesharing service if necessary. Finally, we return the appended trips to the caller (line 7).

```
   Data: source: src, destination: dst, time: departure time,
         number of nearest stops: K
   Result: a set of trips from a source to a destination
   /* Initialize the trips array                          */
1  $trips = []$;
   /* Get the nearest K stops from the source             */
2  $nearby \leftarrow getNearbyStops(src, K)$;
   /* Issue a query to destination starting from each
      stop                                                */
3  for $stop$ in $nearby$ do
   |   /* Calculate the walking from src to the stop
   |      */
4  |   $newTime \leftarrow time + walkingTime(src, stop)$;
   |   /* Calculate the reference trip from stop to
   |      destination starting at the new time            */
5  |   $trips.append(generateReferenceTrip(stop, dst, newTime))$;
   |
6  end
   /* Append a walking distance from the source to
      each nearby stop                                    */
7  $trips \leftarrow appendSrc(trips, src)$;
8  return $trips$;
```

Algorithm 2: Extended Trip Generation Algorithm

3.3 Injecting Ridesharing Services

In this section, we focus on the problem of adding the ridesharing services to the reference trip. The idea here is to iterate over the proposed sub-trips and check if ridesharing can be a better alternative. Since it is complicated to find a random pickup point for a future ridesharing request, we use the fact that the sub-trips start and end at public transportation stops. Hence, we choose the pick-up and drop-off stations among these stops for the service requests. In addition to serving as a good pick-up and drop-off locations, this facilitates the request parameters by knowing the right timing that is derived from the transportation stop's schedule. Therefore, a ridesharing request is issues given the parameters (source, destination and departure time). After a service is found, we connect it to the overall trip. And the gap is filled with the newly found more efficient alternative mode.

It is worth noticing that the added service will improve the arrival time since the new generated sub-plan makes use of the time gained by the ridesharing service. For example, since we will use a service to fill the gap then it is possible that we may arrive in a shorter time. To deal with this case, the planner issues another trip request from the drop-off station to the destination given the new departure time. The new trip will in turn be an input to the core planner for further optimizations.

The proposed algorithm is shown in Algorithm 3. It starts by iterating over the edges of the reference trip, starting from the origin point to the destination. For each edge, the algorithm queries a ridesharing service giving the source and destination points as the departure and arrival stops of the current edge (line 2). Later, the cost of original edge is compared with the cost of adding the new ridesharing service (line 3). In this implementation, the cost function takes the trip time. If the cost is found to be lower, the new trip is replaced with the previous edge (line 4). The addition of the new edge will cause a better arrival time since the cost is lower. Therefore, the algorithm calculates another trip staring from the arrival stop in order to make benefit of the gained time (line 5). Finally, the path originating from the next edge is replaced with the new trip and the algorithm continues iterating over the new edges. When the iteration is completed the algorithm terminates with the reference trip being optimized with the added ridesharing edges.

```
Data: reference trip: trip
Result: Optimized trip with gaps filled
/* Iterate over each edge in the trip */
1 for current, next in trip.getPaths() do
      /* Query for a ridesharing service between the
         endpoints of the edge */
2     $service \leftarrow selectService(current, next)$;
      /* Check if the ridesharing service is faster
         than the existing solution */
3     if (cost(service) ¡ cost(edge(current, next)) then
          /* Replace the existing edge with the
             ridesharing service */
4         replaceEdge(current, next, service);
          /* Issue a new query from the edge endpoint
             to the destination */
5         $newTrip \leftarrow calculateTrip(next, rtrip.dest)$;
          /* Replace the sub-trip from the edge
             endpoint to the destination by the new
             sub-trip */
6         replaceTrip(next, trip.tail, newTrip);
7     end
8 end
```

Algorithm 3: Inject Service Algorithm

3.4 Selecting the Driver/Service

Drivers are moving in real time and finding their future positions is somehow complex. We may use some learning or probabilistic approaches to calculate the future position of a car. However, this is unnecessary due to the fact that in ridesharing services the driver must confirm the request. This means that a candidate driver can simply reject a trip request if it is not along his way.

As emphasized above, a ridesharing service is highly dynamic, which requires to adjust the plan by requesting the ridesharing service in real-time. However, the request should anticipate the future positions of the potential drivers. To overcome this problem, we propose the use of a range query around the arrival stop (pick-up location) so as to select, as candidates for ridesharing, cars that may reach this stop in a specific time in the near future. We send these cars the request with the passenger's waiting time constraint, then we choose the one who accepts it. In case some candidates were chosen but moved away from the stop, the problem can be easily fixed by the driver simply discarding the request.

The range query must take into account two main constraints. The rider pick-up time window constraint and the driver waiting time constraint. The rider pick-up time window constraint is the acceptable interval of time the rider should be picked up at. This is translated into the radius of the range query that covers all the cars that can reach the pick-up position while satisfying the time window. The driver waiting time constraint is the time a driver can wait in the area for the rider to arrive. This is taken into account by first querying the drivers that can reach the rider's pick-up location before the pick-up time window. Then filter them based on the waiting time each driver can wait. The final set of filtered drivers receive the request to be accepted or rejected.

3.5 Triggering the Service Injection

An important question that arises here is where to to inject these ridesharing services? There is no direct answer to this question because it depends on the user preferences. If a user wishes to minimize the travel cost, then it is better to limit the services injection to only the long distance gaps. Ridesharing services usually cost more than public transportation systems, thus limiting their use to only long gaps is helpful. If the user favors optimizing time, then more services are to be injected whenever possible. A balanced approach is to check the overall cost that is composed of both time and money after each service injection. Then select the one with the lowest cost. Another approach is to track the reference trip in real-time, and only inject a ridesharing service when some unexpected delay is detected. In this case, the trip will be flexible to events and adapt in real-time to offer an optimized service to the user.

3.6 Customizing RETRy

It is crucial to note that RETRy is highly customizable and does not depend only on external services and data on the web. Instead, RETRy may work in different configuration based on each need. For example, it may work on local data and local functions provided by the user. To do so, we introduce two functions in algorithms 1 and 3, calculateTrip and selectService. These functions are they key functions behind customizing RETRy. Users may extend those functions to write their own implementation of calculating the trips and selecting the ridesharing functions. Therefore, users are not limited to any service mentioned in the paper and the framework is generic to support different usages.

Adding custom services or functions must be taken with deep caution. The more optimized the algorithm, the better the performance of the framework. Users may rely on quick shortest path algorithms like [8] and lightweight connections representations as in [16].

On the other hand, it is important to note that using external services makes the framework lighter, thus able to be integrated on small devices with limited memory capabilities.

4 EVALUATION

In this section we evaluate the different implementations of RETRy and its core algorithm. We first evaluate the use of a single reference trip as a base plan (as in Algorithm 1), then we study the effect of including the K-nearest trips (as in Algorithm 2). The evaluation criteria is based on comparing different trip plans with and without integrating ridesharing services.

The implementation of RETRy is carried out using Python 3 programming language. The execution environment is a Windows 10 machine with 8GB of RAM and a core i5 processor with 1.70 GHz of processing power.

4.1 Query and Service Selection

The trip queries were formed by randomly choosing different source and destination points. Each query set consists of 200 queries representing random trips in the Ile-de-France region, which is one of the most crowded areas in France. First, we chose a random source and destination for a query. Then, we issue a car driving trip and a multimodal public transportation trip. Later RETRy optimizes the trip by injecting the ridesharing service and finally comparing the results. We have run our queries over one week with a one hour time window each day. The time window choice is to be able to visualize how trips vary with respect to different query hours (morning, mid-day and night).

We chose Google Transit as our trip planning service since it integrates multiple public transportation modes and provides a practical use via an API[1]. Uber[2] was selected as our ridesharing service since the API provides methods to estimate a ridesharing request and get an estimate of both cost and duration. Notice that the algorithm is independent of the services used.

4.2 Single Reference Trip

At first we evaluate the approach by creating a reference trip at a time in each query as in Algorithm 1. In order to optimize the trip, we iterate over each sub-trip and try to inject the ridesharing service in its place. However, since we are injecting the ridesharing service on each edge, we may sometimes consecutive trips using ridesharing modes e.g. trip A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D where the edges between (A and B) and (B and C) are ridesharing services. Therefore, we aggregate the whole sequence by replacing it with one trip from the from the first departure location to the last arrival location (from A to C). This may lead to an earlier arrival, lower cost and thus a more optimized trip.

The mentioned steps were executed on RETRy and the results are shown in figures 1 to figure 4. It is worth to mention that calculating the reference trip and injecting the ridesharing services takes around 8 seconds. Figure 1 shows the trip duration with respect to the hours when the queries were issued. For each hour, a group of different transportation modes are shown with their corresponding values. Analyzing the figure, we notice that the average trip dura-

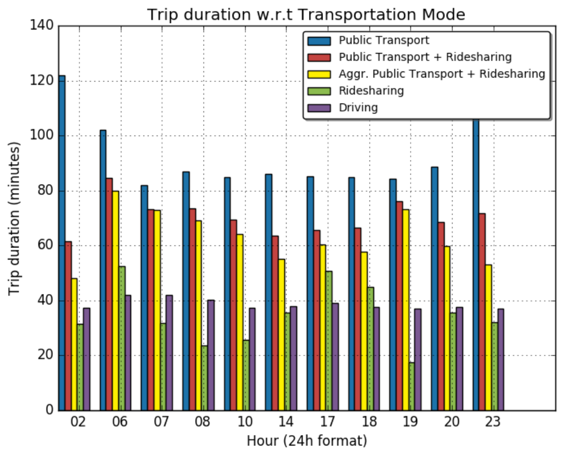


Figure 1: Trip Duration w.r.t Travel Mode

tion (92 minutes) using public transportation is optimized using the injection of ridesharing services (70 minutes) with a gain of approximately 22 minutes in average. This result is better optimized when aggregating the trip (63 minutes) with an average total gain of 29 minutes. We notice that the driving and the ridesharing modes are the fastest according to the simulations (approximately 40 minutes), however, this solution is not feasible for all passengers. The first reason is that not all passengers can travel by car. The second is that using ridesharing services alone is expensive.

In addition to what precedes, we must keep in mind that ridesharing services exist for small distances, therefore relying on them alone is not possible for long trips. Figure 2 shows that the injection of ridesharing trips costs around a minimum estimate of 15 euros which is improved slightly to 11 euros when compressed, while ridesharing alone is nearly double the value (22 euros) which is obviously not preferred.

For practical reason, that is grouping the tests in the same program, the previous experiments considered trips that are planned hours before their execution. However, to ensure the reliability of the proposed solution for timely execution, we set up a simulation script which schedules and calls the ridesharing services at the exact time when they were planned. Then, we compare the planned

[1]https://developers.google.com/maps/documentation/directions/
[2]https://www.uber.com

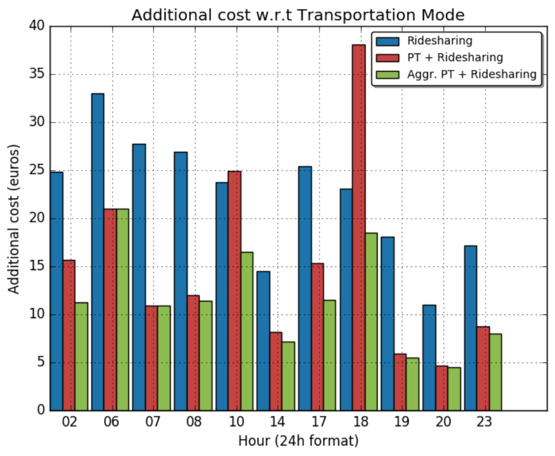


Figure 2: Trip cost w.r.t Travel Mode

arrival time with the real one. The results are shown in Figure 3. The positive points show an early arrival with a max of 16 minutes ahead of time, while the negative points show a late arrival with a max of 7 minutes delay.

Thus, timely trip plan execution is almost as efficient as the offline trip plan, and sometimes more efficient than the offline version. This variation is normal since the real time conditions vary. This test shows however that this variation is rather small, and that our previous experiments are valid. The indirect outcome is to show the stability of the duration between the planned and the online (re)planned trips.

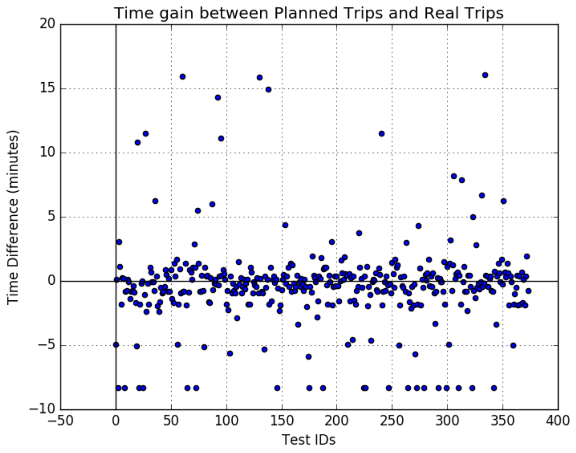


Figure 3: Expected Arrival Time vs Real Arrival Time

4.3 K-Nearest Stops

In this section we study the possibility of improving the results with the querying technique introduced in Algorithm 2. First, we query the nearest K stops (with K = 5), then we calculate trips to destination by taking these stops as starting points. Finally, we add the required path to pass from the starting point to each nearest stop. Despite that this method will clearly result in a worse trip than the reference trip, it is possible that it will result in a optimized trip when ridesharing services are injected.

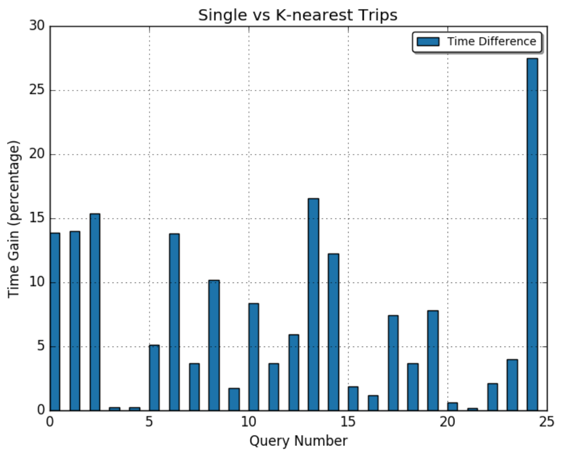


Figure 4: Single vs K-nearest Trips - Gain Percentage

We evaluated this approach by issuing the same queries as in the "Querying and Service Selection" section according to the different proposals in algorithms 1 and 2. The results are shown in Figure 5. The figure represents the durations of the single and k-nearest trips. A clearer view is shown in Figure 4 that presents the time gain by using the K-nearest trips method. The time gain is calculate as follows: $(trip2.duration - trip1.duration) * 100/trip1.duration$, where trip2 is the K-nearest trip and trip1 is the single reference trip. The results show that using the K-nearest stops method (Algorithm 2) will either improve or give the same results as the previous algorithm. The average gain is approximately 8 percent which can be taken as a considerable one especially when adding up the improvement we achieved compared to only using public transportation planners.

However, this improvement comes with a calculation drawback which increases the execution time of the algorithm. Figure 6 shows the execution time in both approaches. On average, the first approach takes 10 seconds to query and optimize the trip, while the second approach takes up-to 60 seconds. This increase is typical since more trips are queried where each one undergoes an optimization phase in order to finally select the fastest one. Nevertheless, we observe that the increase ratio remains reasonable and the execution still doable in near real-time. For instance, for 5 neighbors at both origin and destination w(which results in 25 reference trips), the increase ratio of the execution time is only between 4 and 5.

As a final comparison, Figure 7 shows the difference of using single reference trip and K-nearest trips approaches according to different values of K. The results shows that the duration and cost

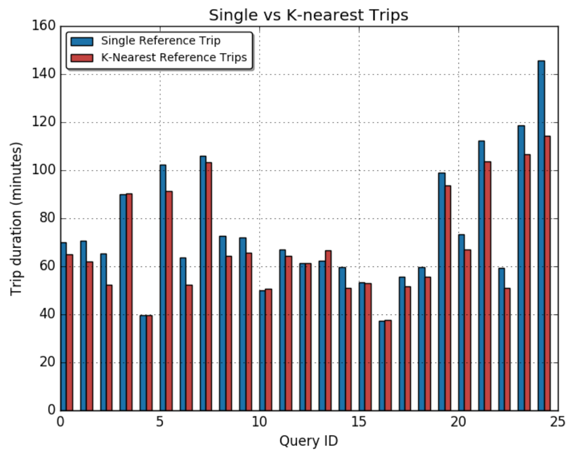


Figure 5: Single vs K-nearest Trips - Trip Duration

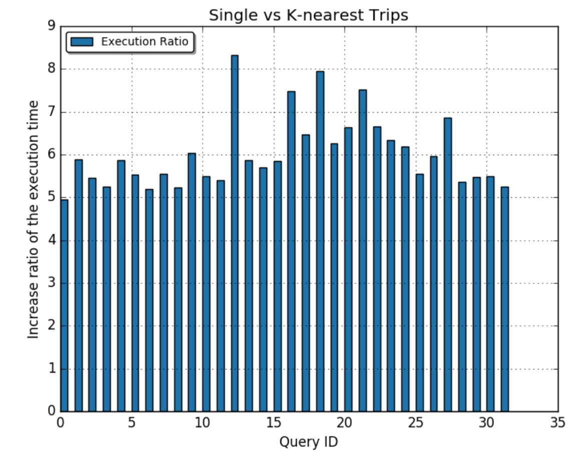


Figure 6: Single vs K-nearest Trips - Execution Time

gain is nearly similar. However, the execution time significantly increases with the increase of K, which is predictable since the number of trip queries is increasing. Therefore, we conclude that K=3 is enough to achieve better trips with a reasonable amount of time.

4.4 Visualization

Our system is split into two main applications. The back-end python core and the front-end Django web application. It allows to interactively query trips and see the results with different choices of transportation modes. Users can see the time and cost difference in using driving, ridesharing, public transportation or our integration solution. A screenshot of the web application is shown in Figure 8, with the first page showing the query interface and the corresponding results in page two. The results are shown as both a table and a graph showing the different durations and costs across the different trip modes i.e. PT, PT + Ridesharing, Aggr. PT + Ridesharing, Ridesharing and Driving. The panel below shows the trip details for each mode where the user may see each trip step in the plan. The map on the right shows the corresponding plot of the trip and is changed when a different mode is selected.

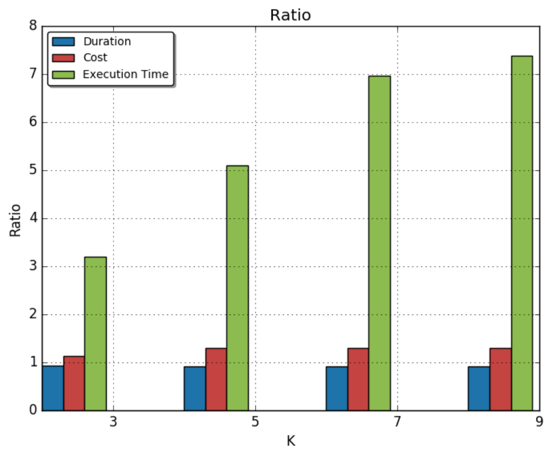


Figure 7: Gain ratio with respect to different values of K

5 CONCLUSION

Ridesharing services play an important role as part of the existing transportation services. Current solutions propose ridesharing as separate services without really integrating them within planned solutions. Our approach presents a light-weight framework that makes use of existing trip planning algorithms and optimizes them by injecting ridesharing services when possible. These characteristics make it possible to be used on mobiles or embedded systems without being overwhelmed with heavy computations. Results show that using ridesharing services in multimodal trip planners will decrease the trip duration with an acceptable extra cost. Future work target enabling multi-criteria querying (such as number of transfers, preferred modes and locations) which is challenging, especially across different services. In addition, we also target integrating a real-time travel monitor with the ability to automatically detect delays or unexpected events and adapt the trip based on them.

REFERENCES

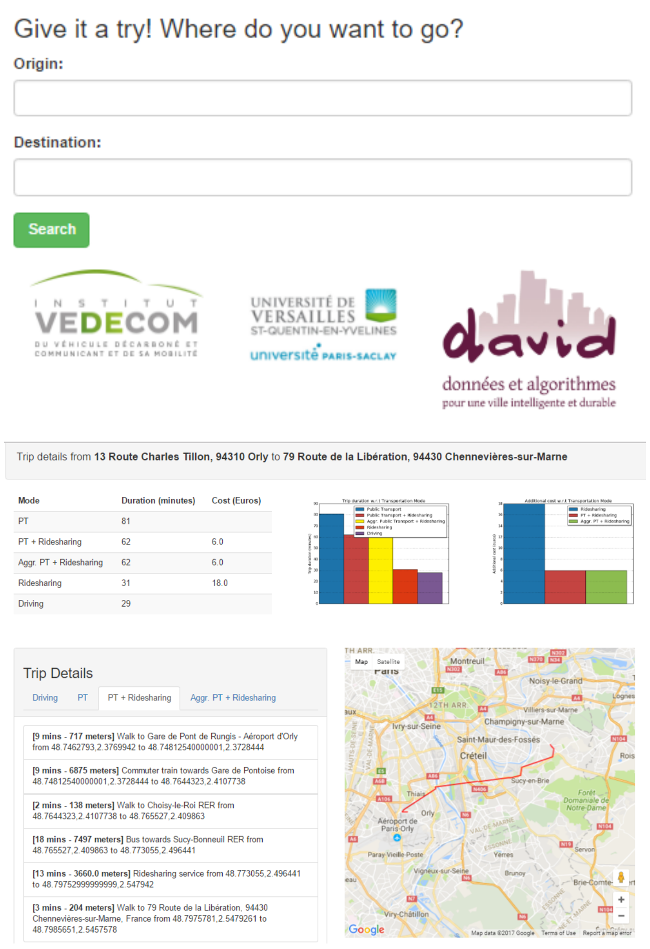
[1] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. 2016. Route planning in transportation networks. In *Algorithm Engineering*. Springer, 19–80.

[2] Richard Bellman. 1958. On a routing problem. *Quarterly of applied mathematics* 16, 1 (1958), 87–90.

[3] Annabell Berger, Martin Grimmer, and Matthias Müller-Hannemann. 2010. Fully dynamic speed-up techniques for multi-criteria shortest path searches in time-dependent networks. In *International Symposium on Experimental Algorithms*. Springer, 35–46.

[4] Bin Cao, Louai Alarabi, Mohamed F Mokbel, and Anas Basalamah. 2015. Sharek: A scalable dynamic ride sharing system. In *Mobile Data Management (MDM), 2015 16th IEEE International Conference on*, Vol. 1. IEEE, 4–13.

[5] Alberto Colorni and Giovanni Righini. 2001. Modeling and optimizing dynamic dial-a-ride problems. *International transactions in operational research* 8, 2 (2001), 155–166.


Give it a try! Where do you want to go?
Origin:
Destination:
Search
INSTITUT VEDECOM DU VÉHICULE DÉCARBONÉ ET COMMUNICANT ET DE SA MOBILITÉ
UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES
université PARIS-SACLAY
david
données et algorithmes pour une ville intelligente et durable
Trip duration w.r.t Transportation Mode
Public Transport
Public Transport + Ridesharing
Aggr. Public Transport + Ridesharing
Ridesharing
Driving
Trip duration (minutes)
Additional cost w.r.t Transportation Mode
Ridesharing
PT + Ridesharing
Aggr. PT + Ridesharing
Additional cost (euros)
Map
Satellite
Paris
Montreuil
Noisy-le-Grand
Ivry-sur-Seine
Champigny-sur-Marne
Saint-Maur-des-Fossés
Créteil
Sucy-en-Brie
Thiais
Orly
Aéroport de Paris-Orly
Paray-Vieille-Poste
Yerres
Brunoy
Vigneux-sur-Seine
Viry-Châtillon
Brie-Comte-Robert
Google
Map data ©2017 Google Terms of Use Report a map error

Trip details from **13 Route Charles Tillon, 94310 Orly** to **79 Route de la Libération, 94430 Chennevières-sur-Marne**

Mode	Duration (minutes)	Cost (Euros)
PT	81	
PT + Ridesharing	62	6.0
Aggr. PT + Ridesharing	62	6.0
Ridesharing	31	18.0
Driving	29	

Trip Details

Driving | PT | PT + Ridesharing | Aggr. PT + Ridesharing

[9 mins - 717 meters] Walk to Gare de Pont de Rungis - Aéroport d'Orly from 48.7462793,2.3769942 to 48.74812540000001,2.3728444

[9 mins - 6875 meters] Commuter train towards Gare de Pontoise from 48.74812540000001,2.3728444 to 48.7644323,2.4107738

[2 mins - 138 meters] Walk to Choisy-le-Roi RER from 48.7644323,2.4107738 to 48.765527,2.409863

[18 mins - 7497 meters] Bus towards Sucy-Bonneuil RER from 48.765527,2.409863 to 48.773055,2.496441

[13 mins - 3660.0 meters] Ridesharing service from 48.773055,2.496441 to 48.79752999999999,2.547942

[3 mins - 204 meters] Walk to 79 Route de la Libération, 94430 Chennevières-sur-Marne, France from 48.7975781,2.5479261 to 48.7985651,2.5457578

Figure 8: A Screenshot of the System

[6] Kenneth L Cooke and Eric Halsey. 1966. The shortest route through a network with time-dependent internodal transit times. *Journal of mathematical analysis and applications* 14, 3 (1966), 493–498.

[7] Daniel Delling, Thomas Pajor, and Renato F Werneck. 2014. Round-based public transit routing. *Transportation Science* 49, 3 (2014), 591–604.

[8] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. 2013. Intriguingly simple and fast transit routing. In *Experimental Algorithms*. Springer, 43–54.

[9] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[10] Robert W Floyd. 1962. Algorithm 97: shortest path. *Commun. ACM* 5, 6 (1962), 345.

[11] Masabumi Furuhata, Maged Dessouky, Fernando Ordóñez, Marc-Etienne Brunet, Xiaoqing Wang, and Sven Koenig. 2013. Ridesharing: The state-of-the-art and future directions. *Transportation Research Part B: Methodological* 57 (2013), 28–46.

[12] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[13] Yan Huang, Favyen Bastani, Ruoming Jin, and Xiaoyang Sean Wang. 2014. Large scale real-time ridesharing with service guarantee on road networks. *Proceedings of the VLDB Endowment* 7, 14 (2014), 2017–2028.

[14] Michael M Kostreva and Malgorzata M Wiecek. 1993. Time dependency in multiple objective dynamic programming. *J. Math. Anal. Appl.* 173, 1 (1993), 289–307.

[15] Shuo Ma, Yu Zheng, and Ouri Wolfson. 2013. T-share: A large-scale dynamic taxi ridesharing service. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 410–421.

[16] Ali Masri, Karine Zeitouni, Zoubida Kedad, and Bertrand Leroy. 2017. An Automatic Matcher and Linker for Transportation Datasets. *ISPRS International Journal of Geo-Information* 6, 1 (2017), 29.

[17] Matthias Müller-Hannemann and Karsten Weihe. 2001. Pareto shortest paths is often feasible in practice. In *International Workshop on Algorithm Engineering*. Springer, 185–197.

[18] Ariel Orda and Raphael Rom. 1990. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM (JACM)* 37, 3 (1990), 607–625.

[19] Ariel Orda and Raphael Rom. 1991. Minimum weight paths in time-dependent networks. *Networks* 21, 3 (1991), 295–319.

[20] Manfred Padberg and Giovanni Rinaldi. 1991. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review* 33, 1 (1991), 60–100.

[21] Frank Schulz, Dorothea Wagner, and Karsten Weihe. 2000. Dijkstra's algorithm on-line: an empirical case study from public railroad transport. *Journal of Experimental Algorithmics (JEA)* 5 (2000), 12.

[22] Bilong Shen, Yan Huang, and Ying Zhao. 2016. Dynamic ridesharing. *SIGSPATIAL Special* 7, 3 (2016), 3–10.

[23] Sascha Witt. 2015. Trip-based public transit routing. In *Algorithms-ESA 2015*. Springer, 1025–1036.

[24] Sascha Witt. 2016. Trip-Based Public Transit Routing Using Condensed Search Trees. *arXiv preprint arXiv:1607.01299* (2016).