Evaluating Ridesharing Algorithms using the Jargo Real-Time Stochastic Simulator

James J. Pan
Tsinghua University, China
pan-j16@mails.tsinghua.edu.cn

Guoliang Li
Tsinghua University, China
liguoliang@mail.tsinghua.edu.cn

Yong Wang
Tsinghua University, China
wangy18@mails.tsingua.edu.cn

ABSTRACT

Ridesharing algorithms operate in environments that are dynamic and uncertain due to traffic effects. Evaluating an algorithm by deploying it in a real environment is costly and often inaccessible, yet the traditional approach of using static inputs and applying an objective function on the outputs may give unrealistic results. Jargo is a novel real-time simulator that provides more realistic evaluation. It lets users implement their own algorithms, speed field functions, and evaluators, and then it reports on multiple quality metrics that are useful to service providers. To support any new and existing algorithm, simulate traffic, and compute the metrics, it is supported by a new relational model of ridesharing. Relations naturally express empirical concepts such as customer pick-up time, and their flexibility can allow any feasible routing strategy. Relational algebra is also convenient for defining operations on the system as well as formalizing service-related metrics. We will show how a service provider considering whether or not to deploy the well-known greedy insertion algorithm could use Jargo to uncover its limits and guide the development of new techniques.

PVLDB Reference Format:
James J. Pan, Guoliang Li, Yong Wang. Evaluating Ridesharing Algorithms using the Jargo Real-Time Stochastic Simulator. *PVLDB*, 13(12): 2905–2908, 2020.
DOI: https://doi.org/10.14778/3415478.3415505

1. INTRODUCTION

In large-scale ridesharing systems, thousands of passenger vehicles move in their own ways through a road network. How they transport customers can be described by their motions, and these motions can be controlled to some degree toward measurable objectives such as maximizing revenue. Service providers seek high-quality *ridesharing algorithms* that can compute optimal motions or determine the optimal customer assignments. Generally, the quality of an optimization algorithm can be measured using static inputs and applying an objective function on the outputs. But for ridesharing algorithms, this approach may give unrealistic results. Under normal operation, inputs to these algorithms are not static but depend on prior outputs and on throughput. For example, a prior assignment can remove a vehicle as a future input due to motion or seating constraints, and a slow throughput can prevent customer requests from getting processed. Outputs themselves can differ from actual motions due to stochastic travel times, invalidating an objective value computed solely on the algorithm's outputs. These effects are hard to quantify in a static setting. Quality could be measured by deploying an algorithm on real vehicles and customers, but this approach is costly and inaccessible to many researchers.

Existing simulators are either black-box with unknown capabilities, unavailable to the public, or do not consider all the effects on inputs and outputs [5, 7, 6, 4, 1]. The difficulty of evaluating these algorithms may contribute to why after a decade of study there are still no widely accepted benchmark instances or results, despite many for the closely related Dial-A-Ride Problem [2].

Jargo is our second attempt at a suitable evaluation tool. It improves on our previous work [5] by including stochastic effects and formalizing many more quality-related metrics that are useful to service providers. Thus it can be used by industrial researchers to assess algorithm quality across realistic workloads and traffic conditions, and it can be used by academia to form a corpus of benchmarking results.

In our demonstration, we will use the New York City 2018–19 New Year's Eve for-hire requests data[1] as our workload. The main feature is a 135% increase in the request arrivals rate within a short span of 5 minutes. For spiky workloads, current providers may use "surge pricing" to discourage requests and reduce workload, potentially at the expense of customer satisfaction [2]. We will show how a service provider can investigate the limits of the popular greedy insertion (GRINS) algorithm [3] for the workload, and under heavy traffic. The results then lead to designing a new fallback mechanism when low throughput is detected and to a new traffic-avoidance strategy.

A New Relational Model of Ridesharing. We introduce a new ridesharing model to overcome the limitations of the graph-based model favored by [8] and others. In the graph-based model, each vehicle is associated to a sequence of customer pick-up and drop-off locations called a schedule, and also to a path through a graph representing the

[1]Obtained from `https://data.cityofnewyork.us/`

[2]`https://hbr.org/2015/12/everyone-hates-ubers-surge-pricing-heres-how-to-fix-it`

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415505

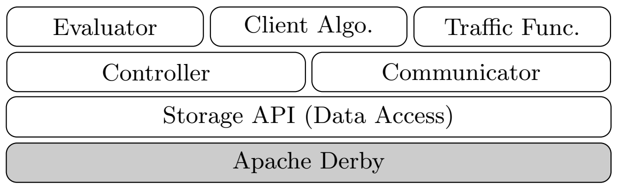


Figure 1: Jargo architecture.

road network. To formalize specific path segments, the path is required to be a shortest path so that elements can be uniquely identified by value. Then, metrics such as customer onboard duration can be formalized by taking the path segment between the pick-up and drop-off locations. Cycles break these formulations because segment endpoints would be unidentifiable. To allow cycles, for example to model a vehicle that circles the airport several times, timestamps on each element would enable relational predicates to be used to define these segments. The model also has difficulty modeling vehicles with different travel durations for the same edges, for example to model a vehicle that waits at the airport compared to one that does not. Again, timestamps added to the vehicle paths would solve this problem.

For wide appeal, a ridesharing simulator should support any feasible routing strategy, including cycles and waits. It should also simulate traffic effects and report useful metrics. Jargo uses relations to support these features. Vehicle paths (routes) are stored into a table with time and position columns to address the above limitations. With this table, traffic is simulated by adjusting values in the time column based on a configurable traffic function. As real-world speed data is often unavailable, such a function can provide synthetic speed values for every road over all time in the simulation, or look up real values if data is available. Relations also facilitate reporting metrics. Route duration, for instance, is simply the range of the time column. For customer-related metrics, a seperate table of pick-up and drop-off times is used. Relational algebra is convenient for formalizing read/write operations and metrics.

Throughout the demonstration, attendees can use the interface to submit investigate queries, monitor the quality metrics, and interact with the live map.

2. OVERVIEW

The ridesharing problem can be formulated as: Given a road network $\mathcal{G}$, a problem instance X, and a traffic function G, produce a *server relation* $\mathcal{X}$ that can optimize an objective function based on service metrics. The purpose of Jargo is to provide simulated real-world conditions under which $\mathcal{X}$ is constructed.

Road Network. Road network $\mathcal{G}$ is modeled as a directed graph, including maximum free-flow speeds along each edge and physical coordinates for each vertex.

Problem Instance. Problem instance X is a listing of customers and vehicles (the ridesharing participants) and their properties. Each participant has a unique identifier, an integer load, a time window, and an origin-destination pair. Customers have positive loads indicating number of required seats while vehicles have negative loads indicating seating capacity. Time windows consist of the "early" time when the participant appears on the network and the latest acceptable "late" time the participant can arrive at its destination. Time windows generalize detour-based constraints used in other works. Origins indicate where on the road network the participant appears, and destinations indicate where the participant desires travel to. For vehicles with no destination of their own such as taxi-like vehicles, Jargo supports an "imaginary" destination.

Traffic Function. Traffic function $G(v_1, v_2, t)$ returns a real number between 0..1 indicating the percentage of maximum speed on edge (v_1, v_2) at time t. If real data is available, G could look up and return the real value. If $G = 1$ for all (v_1, v_2, t), then no stochastic effects are simulated.

Ridesharing Algorithm. The `Client` class can be used to implement both search and join-based algorithms [5].

Runtime Mode. In addition to real-time mode, Jargo can run in sequential mode where the simulation pauses until an algorithm finishes processing customers before continuing. This mode eliminates throughput effects.

2.1 Simulation Results

Server Relation. Server relation $\mathcal{X}$ maps vehicles to times, locations, and pick-up and drop-off events. It can be analyzed using SQL queries against the produced `r_server` view, or by querying other pre-built metric views. For example to count the number of pick-ups and drop-offs for vehicle s, use `select count (Lr) from r_server where sid=`s where column `Lr` contains individual pick-up and drop-off events. Metric views contain an identifier colum and a `val` column with metric value. For example to get the pick-up duration for customer r, use `select val from dur_r_pickup where rid=`r where view `dur_r_pickup` uses a subquery to find the difference of the pick-up time and the request early time. Other metrics are listed in Table 1.

Visualization. A graphical interface is included to visualize customers and vehicles in real time. It also displays running metrics by issuing continuous queries on $\mathcal{X}$. If $\mathcal{G}$ and X are geolocated, GIS software (such as QGIS[3]) could be used for visualization.

2.2 Architecture

Jargo is implemented as a Java library (Figure 1). It provides inheritable base classes `Client` and `Traffic` for developing ridesharing algorithms and traffic functions. It also has an API for developing evaluators. The release package includes example algorithms and traffic functions along with command-line and graphical evaluators. The `Client` and `Traffic` classes use the `Communicator` API to read and write state changes. Evaluators use the `Controller` API to set simulation parameters and display metrics to the end user. Both `Communicator` and `Controller` use the `Storage` API to access the underlying data model. Simulation state is stored in an Apache Derby[4] database.

Workflow. An evaluator calls either `startSequential` or `startRealtime` to start a simulation. The `Controller` reads the reference world time from the problem instance, then it launches four threads (Figure 2). The threads run in parallel if the runtime mode is real-time, otherwise they run sequentially. If real-time, the clock thread advances the world time by one unit every physical second. The request retrieval thread uses `Storage` to retrieve customers appearing on the road network at each world time. It then places them into the `Client`'s job queue. The request handling thread calls `Client.handleRequest` on each of the requests in the queue. The server thread retrieves vehicle locations at each world time and places them into `Client`'s internal memory.

[3]`https://www.qgis.org`

[4]`http://db.apache.org/derby/`

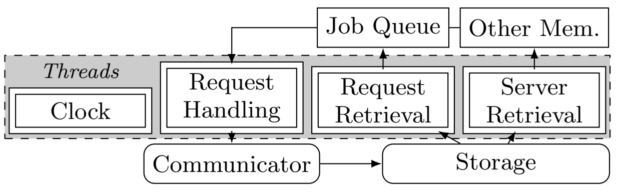


Figure 2: Simulation workflow.

The dynamic components are the *routes and schedules* in the server relation. A *route* is a sequence of *waypoints*. A waypoint is a pair (t_i, v_i) indicating that a vehicle is at location v_i at time t_i. A *schedule* is a route subsequence that lists pick-up and drop-off events, with each waypoint (t_j, v_j) associated to a set of customer labels indicating which customers are being picked up or dropped off. The complete time-evolutions of all vehicle routes are kept. Motion is simulated by sliding over routes to reveal last-visited locations based on the world time. The number of simulated moving vehicles is limited by the read operation with no need for physics-based simulation. Traffic is simulated by adjusting future location timings in the routes based on G.

Through `Communicator`, Jargo controls which data are exposed to ridesharing algorithms. In the offline problem, any update to any part of the routes and schedules can be permitted. But in the online problem, prohibiting updates to the traveled portions of routes and schedules is a more realistic condition. Likewise, only information about previously revealed customers can be used when making routing and scheduling decisions. The SQL schema enforces constraints on routes, schedules, and vehicle capacity.

3. RIDESHARING MODEL

User Relation. Customers and vehicles are called *requests* and *servers* in the model. Their properties are loaded from X and stored in the user relation $\mathcal{U}$, with each row (tuple) representing a single request or server. In the database, $\mathcal{U}$ is split into key-value *property* tables to use in foreign keys.

Server Relation. Server relation $\mathcal{X}$ consists of a server identifier s, a time t, a vertex $v \in \mathcal{G}$, and a set of labels L. Each row indicates that a server was at a specific location at the indicated time, and that the requests in L were picked up or dropped off by that server at the (t, v) waypoint. Set L can be empty if no pick-up or drop-off events occurred.

Constraints. Constraints on $\mathcal{X}$ enforce physical ridesharing constraints, namely: (1) vehicles start and finish service at their origins and destinations, and within certain time windows; (2) vehicles pick-up and drop-off customers at customer origins and destinations, with pick-ups preceding drop-offs, and within certain time windows; (3) a customer is serviced by at most one vehicle; (4) vehicle seating capacity is never exceeded. The rules are implemented by splitting $\mathcal{U}$ and $\mathcal{X}$ into *solution* and *constraint* tables. Solution tables store routes and schedules, and they enforce path integrity of routes and maximum edge speeds. Constraint tables store columns from other tables for cross-table constraints.

3.1 Read Operations

Unassigned Customers. At time t, new requests are retrieved by selecting from $\mathcal{U}$ where t is within the request's time window and where the request is not yet assigned. A single relational equation can formalize this set, but to avoid recomputation we first do $\sigma_{\mathtt{e}\leq t\wedge\mathtt{q}>0}(\mathcal{U})$ to get eligible requests, then compare against a cache of assignments to get the unassigned ones. The e-component is the "early" time and the q-component is the load.

Table 1: Jargo base metrics (s = server, r = request). For definitions see (`https://jargors.github.io`).

S. Travel Dist. $D(\mathcal{X}, s)$	S. Travel Dur. $\delta(\mathcal{X}, s)$
S. Service Dist. $D^{\text{service}}(\mathcal{X}, s)$	S. Service Dur. $\delta^{\text{service}}(\mathcal{X}, s)$
S. Cruising Dist. $D^{\text{cruising}}(\mathcal{X}, s)$	S. Cruising Dur. $\delta^{\text{cruising}}(\mathcal{X}, s)$
R. Transit Dist. $D^{\text{transit}}(\mathcal{X}, r)$	R. Transit Dur. $\delta^{\text{transit}}(\mathcal{X}, r)$
R. Detour Dist. $D^{\text{detour}}(\mathcal{X}, r)$	R. Detour Dur. $\delta^{\text{detour}}(\mathcal{X}, r)$
Service Rate $\mu(\mathcal{X}, t)$	R. Travel Dur. $\delta^{\text{travel}}(\mathcal{X}, r)$
Assigned Requests $\mathcal{R}^{\text{ok}}(\mathcal{X}, t)$	R. Pick-up Dur. $\delta^{\text{pickup}}(\mathcal{X}, r)$
Unassigned Requests $\mathcal{R}^{\text{ko}}(\mathcal{X}, t)$	

Table 2: Demonstration dataset.

Parameter	Value
Road network	Manhattan (31,444 edges, 12,320 vertices)
Problem instance	1,000 vehicles, 22,168 customers
Vehicle capacity	3
Time windows	+6 minutes
Start time	12/31/2018 11:30 PM
Duration	120 minutes

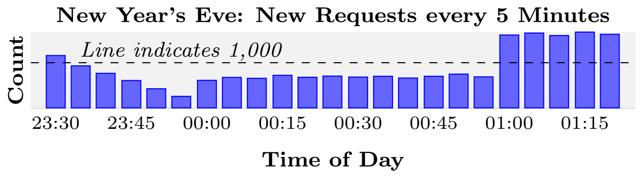


Figure 3: For-hire vehicle trips, 2018–2019 New Year's Eve.

Vehicle Routes and Schedules. The route for server s is obtained by selecting from $\mathcal{X}$ where the s-component equals s and projecting the t and v components, formally $\pi_{\mathtt{t},\mathtt{v}}(\sigma_{\mathtt{s}=s}(\mathcal{X}))$. The schedule is obtained by selecting rows with non-empty label sets, $\pi_{\mathtt{t},\mathtt{v}}(\sigma_{\mathtt{s}=s\wedge|\mathtt{L}|>0}(\mathcal{X}))$.

Vehicle Position. When problem X is loaded, initial routes are computed using G-tree [9] and using maximum speeds on the edges. The last-visited location for s at time t equals $\pi_{\mathtt{v}}((w_{\leq t})_{|w_{\leq t}|})$ where $w_{\leq t} = \pi_{\mathtt{t},\mathtt{v}}(\sigma_{\mathtt{s}=s\wedge\mathtt{t}\leq t}(\mathcal{X}))$ contains the route segment where the time component is no later than t, ordered by time ascending. We set a lower-bound on t during the selection to take advantage of B+ tree indexing.

Quality Metrics. We propose definitions for a variety of real-world service metrics using relational equations on our model (Table 1). For example, for request r with origin $r_{\mathtt{o}}$ and early time $r_{\mathtt{e}}$, the *request pick-up duration* is defined as $\delta^{\text{pickup}}(\mathcal{X}, r) = \pi_{\mathtt{t}}(\sigma_{\mathtt{v}=r_{\mathtt{o}}\wedge r\in\mathtt{L}}(\mathcal{X})) - r_{\mathtt{e}}$

3.2 Write Operations

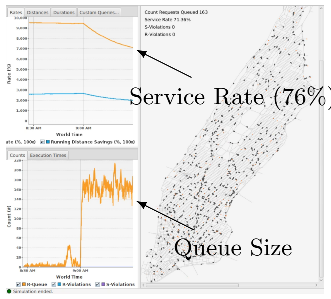
Route Update. To update a route, we delete the untraveled portions $w_{>t} = \pi_{\mathtt{t},\mathtt{v}}(\sigma_{\mathtt{s}=s\wedge\mathtt{t}>t}(\mathcal{X}))$ from a server's route and replace it with the new updated segment.

Traffic Simulation. Before any route update hits the data tables, it is modified for traffic effects. We scan it from beginning to end, and for each adjacent (t_{i-1}, v_{i-1}) and (t_i, v_i) pair, the speed multiplier $G(v_{i-1}, v_i, t_{i-1})$ is computed. Then, t_i is modified so that $d(v_{i-1}, v_i)/(t_i - t_{i-1})$ is within the adjusted flow speed, where d is the distance along the edge. This adjustment is propogated to the later waypoints. The complexity is $O(n^2)$ for an n-length route, and we only need to perform the procedure when the route is inserted or updated.

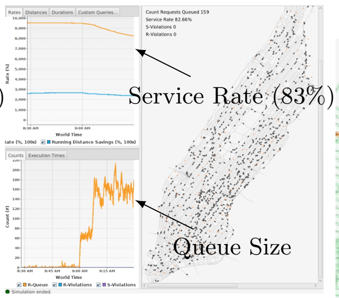
4. DEMONSTRATION

A service provider wants to avoid surge pricing on upcoming New Year's Eve. We will estimate the revenue potential (F^{rev}) for GRINS while also measuring the customer inconvenience (F^{inc}) and environmental impact (F^{env}). We formulate these objectives using Table 1 metrics:

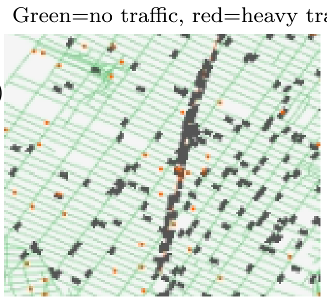
- $F^{\text{rev}}(\mathcal{X}) = p_{\text{base}}|\mathcal{R}^{\text{ok}}(\mathcal{X}, \infty)| + p_{\text{km}} \sum_{s\in\mathcal{S}} D^{\text{service}}(\mathcal{X}, s)$,
- $F^{\text{inc}}(\mathcal{X}) = \sum_{r\in\mathcal{R}^{\text{ok}}(\mathcal{X},\infty)} \delta^{\text{travel}}(\mathcal{X}, r) + p_{\text{ko}}|\mathcal{R}^{\text{ko}}(\mathcal{X}, \infty)|$,



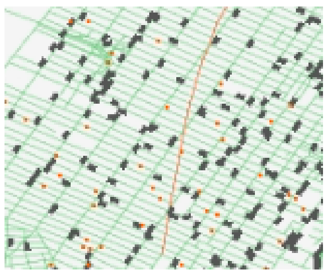
(a) GRINS
No Traffic



(b) GRINS-F
No Traffic



(c) GRINS-F
Broadway Traffic



(d) GRINS-FB
Broadway Traffic

Figure 4: Jargo usage scenario.

- $F^{\text{env}}(\mathcal{X}) = p_{\text{smog}} \sum_{s \in \mathcal{S}} D(\mathcal{X}, s)$.

For F^{rev}, we multiply a base price p_{base} by number of assigned request, plus a price-per-kilometer p_{km} for the total in-service distance traveled by the vehicles. A vehicle is "in service" if it has customers onboard. To get assigned requests, we use $\mathcal{R}^{\text{ok}}(\mathcal{X}, t)$ that counts up requests in $\mathcal{X}$ that are in exactly two labels, indicating they were serviced. We use ∞ for the second parameter to say we want the final count. For F^{inc}, we sum the total request travel duration plus a penalty p_{ko} for each unassigned request. The request travel duration includes time until pickup in addition to time spent in a vehicle. For F^{env}, we multiply a "smog" factor p_{smog} by the total distance traveled by the vehicles. Based on current estimates, we set $p_{\text{base}} = \$2.50$, $p_{\text{km}} = \$1.56$, $p_{\text{ko}} = 15$ minutes, and $p_{\text{smog}} = 0.374$ grams of hydrocarbons.

We will use historical New York City for-hire trips during 2018–2019 New Year's Eve (Table 2) as an example workload. Figure 3 shows the number of new customer requests every 5 minutes, starting from 11:30 PM on December 31, 2018. A dip is seen near midnight (00:00) followed by a surge an hour later (01:00). We generate 1,000 synthetic vehicles with 3-capacity. We initialize the speeds to 10 meters per second (36 kilometers per hour) for all roads.

To establish a baseline, we evaluate GRINS using real-time mode without traffic and find that it achieves 76% service rate (Figure 4a), meaning that a quarter of the customers are not serviced. The algorithm struggles to clear the jobs queue, inspiring a fallback mechanism. The new variation GRINS-F reverts to a fast nearest-neighbor strategy if the queue size exceeds 60. We find GRINS-F can achieve 83% service rate (Figure 4b), an improvement of +9%. New techniques may be needed for further improvement. Now as Broadway is a popular road, we want to see the effect when speed along it is reduced to 20%. The service rate drops back to 76% for GRINS-F, and visual inspection shows many vehicles stuck along Broadway (Figure 4c). This result inspires an avoidance strategy. Variation GRINS-FB adds a large routing cost to Broadway in order to avoid it. We find GRINS-FB can restore the service rate to 83%. The server relation is available for offline analysis after each run, and we use it to compute the final objectives shown in Figure 5. Notably GRINS-FB can achieve +7.7% more revenue compared to baseline GRINS, even when traffic is present. Additionally, revenue seems to be directly proportional to environmental impact while inversely proportional to customer inconvenience. Analyzing these and other relationships can help service providers determine which algorithms to deploy in which scenarios, for example reducing environmental impact or reducing customer inconvenience.

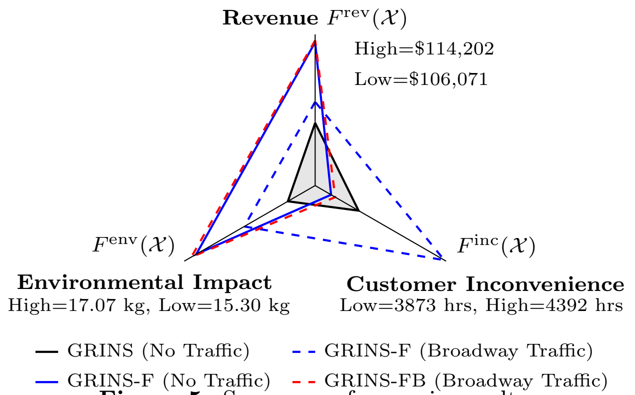


Figure 5: Summary of scenario results.

Acknowledgement. This paper was supported by NSF of China (61925205, 61632016), Huawei, and TAL education.

5. REFERENCES

[1] K. W. A. A. Horni, K. Nagel. *The Multi-Agent Transport Simulation MATSim*. Ubiquity Press, 2016.

[2] J.-F. Cordeau. A Branch-and-Cut Algorithm for the Dial-a-Ride Problem. *Operations Research*, 54(3):573–586, 2006.

[3] S. Ma, Y. Zheng, and O. Wolfson. Real-time city-scale taxi ridesharing. *IEEE Trans. Knowl. Data Eng.*, 27(7):1782–1795, 2015.

[4] M. Ota, H. T. Vo, C. T. Silva, and J. Freire. Stars: Simulating taxi ride sharing at scale. *IEEE Trans. Big Data*, 3(3):349–361, 2017.

[5] J. Pan, G. Li, and J. Hu. Ridesharing: Simulator, benchmark, and evaluation. *PVLDB*, 12(10):1085–1098, 2019.

[6] E. F. S. Hörl, C. Ruch. Amodeus, a simulation-based testbed for autonomous mobility-on-demand l-systems. In *IEEE International Conference on Intelligent Transportation Systems*, 2018.

[7] Z. Xu, Z. Li, Q. Guan, D. Zhang, Q. Li, J. Nan, C. Liu, W. Bian, and J. Ye. Large-scale order dispatch in on-demand ride-hailing platforms: A learning and planning approach. In *KDD*, pages 905–913, 2018.

[8] L. Zheng, L. Chen, and J. Ye. Order dispatch in price-aware ridesharing. *PVLDB*, 11(8):853–865, 2018.

[9] R. Zhong, G. Li, K. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. *IEEE Trans. Knowl. Data Eng.*, 27(8):2175–2189, 2015.