Emerging Cutting-Edge Applied Research and Development in Intelligent Traffic and Transportation Systems, M. Shafik (Ed.)

doi:10.3233/ATDE241175

Dispatching Taxi Cabs with Ridesharing – An Efficient Implementation of Popular Techniques

Bogusz JELINSKI
Skatteetaten[1]

Abstract. The article describes how to effectively dispatch hundreds of thousands of ride requests per hour, with thousands of cabs. Not only which cab should pick up which passenger, but which passengers should share a ride and what is the best pick-up and drop-off order. An automatic dispatching process has been implemented to verify feasibility of such cab sharing solution, simulation was used to check quality of routes. Performance of different programming tools and frameworks has been tested. Thousands of passengers per minute could be dispatched with basic algorithms and simple hardware and they can be dispatched in a cab sharing scheme very effectively, at least 11 passengers per cab per hour. The spotlight is on practical aspects, not well-known theory. The goal is to verify feasibility of a large-scale dispatcher and to give its benchmark. Implementation of algorithms including a dispatcher and simulation environment is available as open source on GitHub.

Keywords. dispatching, minibus, pool, ride sharing, taxi routing

1. Introduction

At least a dozen companies have been working on vehicle autonomous driving with the intention to make urban collective transportation more appealing to car commuters. Taxi services are too expensive, line buses are either uncomfortable or slow. But self-driving capability, albeit most scientifically challenging, is not the only software component we need – we need to assign passengers to cabs or minibuses (later referred to as 'buses'), and, if feasible, we need to assign many passengers to one bus in such a way that passengers' time expectations are met.

There have been numerous studies covering optimization of taxi services, an extensive overview is given by [1-3]. The approach described below is not a classic Vehicle Routing Problem with Time Windows [4] as buses do not have to return to a central depot. It is sometimes called dynamic vehicle routing problem or dynamic ride-matching problem with time windows (DVRP or RMPTW respectively, [5]). Furthermore, most research papers present solutions that "are limited to solve instances of small to medium-sized VRPTW" [2] and "there is no large-scale benchmark for dynamic vehicle routing" [6]. This benchmark is given below – hundred thousand passengers per hour, sixteen thousand buses. Such volume will be needed if city transport is to be more personalized, better suited for real demand, especially when self-driving small capsule-like buses will overtake current timetables.

[1] Corresponding author, E-mail: bogusz.jelinski@skatteetaten.no

To serve such a volume a different computational approach had to be used – no model solver like Gurobi, starting with simpler assignment model instead solved by Munkres method (aka Kuhn-Munkres, Hungarian [7]), supplemented by dynamic programming executed in multithreaded environment. The purpose is to check feasibility of this approach and compare performance indicators with values reported by other researchers. The use of assignment model is justified by industry standards – low-cost method (called also nearest vehicle dispatch, see [5]) is still the most common dispatch strategy.

Randomized data, both trip requests and bus locations, were used in simulations as data from the field with such scale were unavailable to the author. There are at least two premises of such approach – the current taxi movement pattern, which could be described as "occasional", will most likely not apply to a massive ride sharing system serving as the main last-mile transport. Secondly, randomized data with no repetitions is a much-needed challenge for ride-sharing dispatchers. Randomized data can be stored in a file so that simulations are repeatable with different parameter values and different stages being activated (e.g. skipping pool extender). As results are stored in a database, SQL scripts are available to calculate performance indicators and other statistics. While random requests get automatically assigned it is possible to submit a request manually and observe its consequences both for the passenger, the assigned bus and even stops involved.

What most other studies overlook is the meta-optimization part – how the time of finding the optimal or suboptimal solution affects performance indicators. To verify this a real-life infrastructure has been built, with clients calling RestAPI, the state of all objects is stored in a database. Because it has been implemented using open-source tools and is published as open-source with manuals on GitHub[2] it has one more advantage over most other studies – results are easily reproducible.

2. Assignment problem

This classic assignment problem [8] seems to be a challenge at first glance – the complete decision set is a permutation, which implies n! size in a balanced transportation scenario, n*n variables and 2*n constraints. It is a binary integer linear problem with a cost matrix and supply and demand constraints. The expected solution is a plan, buses assigned to requests, that minimizes the total distance/cost to pick up a passenger. We can think of other goal functions e.g. minimizing the total time which customers have to wait or total distance to go. The last one could be reasonable under heavy load to support multimodal urban transport. The cost matrix is constructed based on information about distances between pick-up points and current location of all buses, in kilometers or time it would take to transfer. In advanced scenarios average travel time can depend on the time of day, it can vary due to rush hours.

The model has the following form:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \rightarrow MIN \quad (1)$$

[2] https://github.com/boguszjelinski/kern

$$\sum_{i=1}^{n} x_{ij} = 1 \; for \; j = 1, \ldots, n \tag{2}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \; for \; i = 1, \ldots, n \tag{3}$$

$$x_{ij} \in \{0,1\} \; for \; i, j = 1, \ldots, n \tag{4}$$

3. GLPK solver, Munkres and LCM

Three tools to solve this model have been tested – GLPK solver, Munkres and the least cost method [9]. GLPK solver can give the flexibility to extend constraints and it helped in verification of Munkres algorithm implementations. GLPK program executes surprisingly fast – with n=500 there are about 10^{1134} possible assignments while the number of atoms in observable universe is estimated at 10^{82}, but it takes only about 15 seconds with 4th generation Intel i7 processor to find the optimal assignment. The trouble is there can be thousands of buses available, the time gets doubled with every hundred buses, see Table 1. So, we need a faster method, Munkres comes to the rescue, an open-source implementation needs less than two seconds with n=500 (balanced model). Two different implementations have been tested, one implemented in C and one in Rust.

Table 1. Solver execution, balanced models (time in seconds)

n	GLPK	Munkres/C	Munkres/Rust
100	0.2	0.0	0.0
200	0.8	0.3	0.1
300	2.9	1.1	0.7
400	8.2	2.9	1.3
500	15.5	5.9	2.0

Rust language is considered to be slower than C, but interestingly Rust implementation is faster. LCM is the fastest method, but the speed comes at a price – 300 simulations with n=200 (buses and passengers randomly scattered in Budapest) showed that simple LCM gives at least 30% bigger assignment "cost", 45% on average, which is huge. Both fuel and customers' wait time can be saved with a solver, potentially. This is in line with other research, which states that "optimal solutions to the offline taxi routing problem are usually significantly superior to the output of common local-improvement and greedy routing algorithms" [6].

Worth noticing is that most of taxi dispatching software only uses LCM, new greedy algorithms are being developed [10]. We still would need LCM in two cases – models too big to be given to Munkres, they need to be squeezed by LCM "presolver", and we need LCM in pool finder (see chapter below). When requests are dispatched immediately on arrival, this is also a LCM scenario.

Pool finder and route extender described below are two steps prior to solver stage, which can significantly reduce the size of the assignment model. In fact, during simulations the demand side of the model was small (below 100), but 16000 buses were available, models were highly unbalanced. It has been found that not all Munkres implementations give optimal solution in such case. All modules have been implemented in Rust.

4. Route extender

The very first thing we can do with an incoming transportation request is to try to allocate it to an existing route if matching perfectly or non-perfectly. This operation is sometimes called "insertion" [11]. If it does not match perfectly (one or two stops must be added) we have to check constraints of all other co-passengers. There might be more than one pool that this request could fit in, we can have different goal functions here, for example:

- minimum detour (often used in different implementations – see [12])
- minimum negative impact on duration of the whole pool (this one was used in simulations)
- minimum total detour of all passengers

A full-scan method has been implemented with multi-threading, which is fast enough to dispatch hundreds of thousands requests per hour on a laptop, see Table 5.

5. Pool finder

A pool finder is a method here to find unassigned trip requests that can share a bus. A pool is a shared route which satisfies passengers expectations – wait time and trip duration (see also [13]). A route with three passengers can have from two to six stops, two extreme examples:

1 in, 2 in, 3 in → 1 out, 2 out, 3 out
1 in → 2 in → 1 out → 3 in → 3 out → 2 out

An effective way to search through such trees is dynamic programming [14]. In short you do not recurse over nodes all over again, but you divide search in stages starting from the leaves. At the root level all feasible pools are sorted by total time travel (one of many possible goal functions), repeats are removed. By repeats I mean plans that have a participant which appears in a better plan. The next step would be to assign the first "in" passenger to a bus by the solver, but the assignment must meet requirements of all pool participants, not just the first one in pool's sequence. The assignment must be done in the "pool finder" method, in the loop where repeats are removed. LCM must be used as this is not a simple assignment problem and the model would be huge. Unassigned buses and unassigned customers are given to the next dispatching phase, see below the sequence.

One assumption needs to be emphasized, distances between bus stops must be known in advance, this database can be initialized with geographical distance and updated later with real data from the field.

Performance of this approach is satisfactory – a solution for 200 requests with four seats (passengers in a route, up to 8 stops), which means a tree with 1019 paths/leaves, is found after 50 seconds with and Intel i7 class processor (4 threads). Table 2 shows performance of pool finder implemented in C – three and four passengers, wait time 10min, acceptable pool loss 90% of a lone trip (detour).

Table 2. Impact of number of requests and seats in the bus on performance of pool finder, in seconds

Number of requests	Three seats	Four seats
100	0	2
200	1	50
300	3	-
400	9	-
500	20	-

Because of the calculation time growing exponentially there are some limits in the dispatcher. Limits are customizable depending on the hardware performance, they are intended to protect against requests piling up while the dispatcher is working. The most important limit is how many trip requests can be sent to pool finder, which was executed four times per minute. There are different limits for scenarios with four, three or two passengers as they perform differently. During simulations the limit for four passengers was 120, 440 for three passengers, 800 for two.

If executed several times per minute this pool finder followed by two-step solver stack (LCM presolver supplemented by Munkres, responsible for assignment of trip requests which cannot be shared in a pool) can assign at least one hundred thousand requests per hour while executed in a modern desktop-class processor.

With this step the total dispatching process looks like this:

Route extender (full scan)
↓
Pool finder (dynamic programming & LCM)
↓
Presolver (LCM)
↓
Solver (Munkres)

6. Quality of routes

Simple time constraints can lead to strange routes. A bus will omit a stop to pick-up a passenger with wait time running out and come back to a previous one and then maybe again go to the same next one to drop-off the second passenger, who does not have tight wait requirements. One solution that has been tested is to promote more straight routes for nearby stops, with stop bearings. Regardless of how routes are constructed we should measure the quality of routes, which could help compare different dispatchers and monitor real systems ([3] p. 21, [15] p. 10, [2] p. 13). One can think of metrics presented in Table 3. Of course, passengers and taxi operators have different perspectives and might have different view on a particular metric – more passengers in a pool means more income but a longer and less comfortable trip.

Table 3. Route quality metrics

Metric	Rationale
average wait (pick-up) time	Customers can choose competitors if too long
average trip duration	Affected by detour and quality of dispatcher
average pool size (passengers per route)	The bigger the pool the fewer buses are needed
average passenger count per route's leg	How many seats do buses need to have?
average number of legs	How fragmented are routes? Stops cost time
average detour	How much do customers lose on ride sharing

7. Simulations

There might be different dispatching algorithms, goal functions and constraints. They might be implemented using different tools and frameworks. There might be different ideas about communication protocols, application programming interfaces (API or REST API) and service flows, charging models that could impact total revenue. Therefore, it is desirable to have a simulation environment where behavior of all counterparts and system performance could be tested. Such environment has been implemented – simulators of customers, buses and stops (see Figure 1), supplemented by API and database storage. Two environments actually – one with API and one with direct access to the database for dispatcher tests only. Have a look at Tables 4 and 5 how route extender, pool finder and number of requests have affected results. Results are divided in three sections – how many buses were needed (two first rows), quality of routes (next seven rows) and dispatcher's performance (last four).

Table 4. Comparison of performance with and without route extender and pool finder, 20k requests, 4k buses

Metric	No sharing	Extender only	Pool only	Ext. & pool
total buses used (assigned at least once)	3729	1854	3571	1883
max total buses assigned at a time (peak)	2310	1810	2640	1863
km driven in total	55123	39195	58528	40191
average wait (pick-up) time – min:secs	1:27	10:39	6:56	10:49
average trip duration– min:secs	3:58	4:54	4:05	4:46
average number of passengers per route	1	10.8	2.3	10.6
average passenger count per route's leg	0.57	1.07	0.69	1.05
average number of legs per route	1.76	19.9	4.28	19.5
average detour.	0%	21%	1%	18%
max dispatcher time (seconds)	3	0	17	11
average dispatcher time (seconds)	0	0	3	0
average pool finder input size (demand)	0	0	98	21
average solver input size (demand)	379	31	19	5

Table 5. Comparison of performance with and without route extender, 100k requests, 16k buses

Metric	No sharing	Extender only
total buses used (assigned at least once)	14763	9028
max total buses assigned at a time (peak)	10095	9026
km driven in total	220722	160224
average wait (pick-up) time – min:secs	1:26	10:53
average trip duration– min:secs	4:01	5:44
average number of passengers per route	1	11.07
average passenger count per route's leg	0.66	1.38
average number of legs per route	1.5	17.9
average detour.	0%	42%
max dispatcher time (seconds)	21	14
average dispatcher time (seconds)	2	6
average pool finder input size (demand)	411	46
average solver input size (demand)	14763	9028

Table 4 concerns twenty thousand requests per hour served by four thousand buses. Table 5 shows results with one hundred thousand passengers served by sixteen thousand buses. In all scenarios requests were sent for 60 minutes, distributed evenly over time and randomly over all stops, there were no two identical requests, in terms of from-to stands. "No sharing" means no root extender or pool finder. "Extender only" means that the pool finder step was skipped during simulation. Pool finder was skipped in the bigger

scenario not only due to lack of required computing power – pool finder showed little gain in the smaller scenario.

All requests had the same requirements (although the dispatcher handles individual constraints – each customer can set their own requirements) - maximum waiting time 15min, as in [5], maximum detour 70% ([5] chose 100%), maximum duration of trip without detour 4 min, which should be long enough for last-mile commute. Buses were traveling at 30km/h, there were 5193 stops. Additional logic of the extender and its execution time, which was negligible in any scenario below 50000 requests per hour, decreased total dispatcher execution time significantly, as the size of subsequent, time-consuming steps decreased.

Dispatcher was run every 15 seconds. Some implementations execute dispatcher at every trip request (first-come first-served, see [5]), some do it twice per minute ([6], [16]).

The most eye-catching outcome is that pool finder gives worse results than solver-only solution with no ridesharing. That might be attributed to LCM used in pool finder and the fact, that pool finder gives a lot of empty pick-up legs (with no passenger; a leg is a travel from stop to stop). However, route extender utilizes buses more efficiently. One bus transported 11 passengers per hour on average, and the value can be bigger by limiting the length of requested distance and with the less random nature of commuters' streams. This result, ratio request per bus, is more than twice as good as achieved e.g. by [15], which was 6.

Pool finder without extender gave better wait and detour time than with extender but needed a lot more buses. The reason was the choice of goal function in extender, which was minimizing negative impact on duration of the whole route being extended, not detour of the particular passenger. The same goal was chosen by [15] (p. 3).

Number of legs of a route generated by extender is significantly larger than in pool finder scenario due to the nature of how legs are added to a route, which is not as important as number of passengers per leg. It does not mean degradation of passenger's satisfaction, there was a limit of active legs (10). Route is just a technical aspect of the simulation, a database artefact.

Table 6 shows that extender's efficient bus usage was accomplished more by proper constructing of routes than by occupying many seats at a time (nearly one passenger per leg on average). In the scenario with twenty thousand requests and no pool finder there was nearly no need for three seats or more, two seats were occupied in roughly every fourth leg. It is worth noting that the implementation allows each bus to have its own individual seat constraint.

Table 6. Number of occupied seats in routes' legs – extender only scenario, 20k requests

Number of passengers	Number of legs
0	8045
1	18646
2	10030
3	302
4	2

The above results were achieved on a laptop with i9-9880 mobile processor and dispatcher running on ten threads, sharing computing power with the database. Today's processors have twice as fast cores, and twice as many cores. Several hundred thousand trip requests per hour could easily be served. The only limiting factor in simulation in

extender-only scenario was the performance of the database running on the same hardware.

Figure 1. Visualization of traffic at a stop during simulation (Budapest)

8. Even more passengers

There are some techniques and functionalities that may help put more passengers into a bus. These techniques do not load the dispatcher. Firstly, passengers can be allowed to take co-passengers - one trip request for more than one passenger. Secondly one could join an already defined route at the bus stop - commuters can see details of the route displayed on the approaching bus, on the route tables at the stops or on their own smart phones ("show coming buses"), they approach the bus, authenticate and choose one of the available stops in the route.

It is imaginable that combination of these ideas could lead to a software solution that would be able to dispatch several hundred thousand passengers per hour. How many buses would be needed depends on many factors – among others the allowed travel time and speed of the buses. Twenty passengers per bus per hour seems to be a restrained assessment of an average bus usage.

9. Technology stack

The choice of proper tools turned out to be vital to achieve large-scale results. At the beginning all modules were written in Java (backend) and TypeScript (frontend applications). Pool finder is now also implemented in Rust, C# and C. Java and C#, including multithreading, are at least three times slower than C. C is still faster than Rust, all simulation results presented here have been achieved with pool finder implemented in C. Surprisingly the executable produced by MinGW/MSYS2 gcc compiler v. 11.2.0 runs about two times faster than the same version from Cygwin environment. Java is memory intensive while running thousands of threads, routines simulating customers and buses have been rewritten in Golang (API client) and Rust (direct database access). Java is not the fastest API backend framework either, Rust is used now. Python is used for support scripts, but Rust and MySQL database are the only required components to run simulations, other languages are optional.

10. A comment on driverless buses

We will probably not be able to get rid of the bus driver in near future because:

- most legislatures will not allow for self-driving buses.
- or will limit their speed to a ridiculous level.
- we might need someone in place to reduce fraud, misuse (going in without an order or with an unregistered co-passenger, ordering a shorter trip but staying in for a longer one), vandalism or sabotage, like the one done with traffic cones against Cruise and Waymo.
- driver's presence can improve passengers' comfort and security. Most of us are still afraid of the lack of a driver or can feel discomfort originating from personal issues [13] e.g. "*some male passengers 'interfering' with female passengers*" [17].
- assistance might be needed to support disabled, inexperienced or digitally weak customers.

11. Summary

It has been shown that it is feasible to dispatch one hundred thousand trip requests using an aged laptop. There is no technology enabler missing, it does not require massive computing power. But the choice of technologies used to implement algorithms must be careful, there is a vast difference in performance, garbage-collecting languages have their overhead. A benchmark with widely used constraints and performance indicators is given for further studies on large-scale scenarios. With automatic dispatch algorithms we can significantly increase efficiency of bus service, bus usage ratio (passengers transported within an hour by one bus) can be 10-20, not 3-4 like in New York without ride sharing. Result achieved during simulations are about twice as good as presented by other researchers and they were achieved under unfavorable conditions - with randomly scattered demand without duplicated requests. But passengers' streams are more concentrated around some main routes and transport hubs, especially during rush hours, which means we can achieve much better results in real life.

References

[1] Bräysy O, Gendreau M. Vehicle Routing Problem with Time Windows, Part I: Route Construction and Local Search Algorithms. Transportation Science. 2005 Feb; 39(1):104-118
[2] Liu X, Chen YL, Por LY, Ku CS. A Systematic Literature Review of Vehicle Routing Problems with Time Windows. Sustainability. 2023; 15(15):12004
[3] Maciejewski M, Nagel K. Simulation and Dynamic Optimization of Taxi Services in MATSim. Transportation Science. 2013;10-13
[4] Truden Ch, Maier K, Armbrust Ph. Decomposition of the vehicle routing problem with time windows on the time dimension. Transportation Research Procedia. 2022; 62:131–138
[5] Jaeyoung J, Jayakrishnan R. Dynamic Shared-Taxi Dispatch Algorithm with Hybrid-Simulated Annealing. Computer-aided Civil and Infrastructure Engineering. 2016 Apr; 31(4):275-291
[6] Bertsimas D, Jaillet P, Martin S. Online Vehicle Routing: The Edge of Optimization in Large-Scale Applications. Operations Research. 2019; 67(1): ii-iv
[7] Kuhn HW. The Hungarian method for the assignment problem. Naval research logistics quarterly. 1955; 2:83-97

[8] Wagner HM. Principles of Operations Research with Applications to Managerial Decisions. 2nd ed. New Jersey: Prentice – Hall; 1969. 1095 p.
[9] Taha HA. Operations Research: An Introduction. 10th ed. Pearson; 2017. 843 p.
[10] Kim Y, Young Y. Zone-Agnostic Greedy Taxi Dispatch Algorithm Based on Contextual Matching Matrix for Efficient Maximization of Revenue and Profit. Electronics. 2021; 10(21):2653
[11] Jiachuan W, Peng C, Libin Z, Chao F, Lei C, Xuemin L, Zheng W. Demand-aware route planning for shared mobility services. Proceedings of the VLDB Endowment. 2020 March; 13(7):979-991
[12] Shankar N, Venkatesh K. Real Time Taxi Ride Sharing System. International Journal of Advanced Research Trends in Engineering and Technology. 2016 June; 3(6)
[13] Lalos P, Korres A, Datsikas CK, Tombras GS, Peppas K. A Framework for Dynamic Car and Taxi Pools with the Use of Positioning Systems. Conference: Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns; 2009 Athens; p. 385-391
[14] Bellman RE. Dynamic Programming. New Jersey: Princeton University Press; 1957. 340 p.
[15] Ma OW, Zheng Y. Real-time city-scale taxi ridesharing. IEEE Transactions on Knowledge Discovery and Data Engineering. 2015; 27:1782–1795
[16] Alonso-Mora J, Wallar A, Rus D. Predictive routing for autonomous mobility-on-demand systems with ride-sharing. 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS); Vancouver, BC, Canada, 2017; p. 3583-3590
[17] Enoch M, Potter S, Parkhurst G, Smith M. Why do demand responsive transport systems fail? Transportation Research Board 85th Annual Meeting, 22-26 January 2006; Washington DC