Dispatching taxi cabs with passenger pool – an efficient implementation of popular techniques

Bogusz Jelinski
Mo i Rana, Norway
ORCID: 0000-0002-8018-4908

***Abstract*— The focus of making collective urban transportation more competitive and attractive is on self-driving capabilities. But do we have tools to effectively dispatch hundreds of thousands of cabs or minibuses, thousands of trip requests per minute? Not only which cab should pick up which passenger, but which passengers should share a trip 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. The spotlight is on practical aspects, not well-known theory. Hundred thousand passengers per hour could be dispatched with basic algorithms and simple hardware and they can be dispatched in a cab sharing scheme very effectively, at least 15 passengers per cab per hour. Implementation of algorithms including a dispatcher and simulation environment is available as open source on Github.**

Keywords — dispatching, minibus, pool, sharing, taxi,

Key take-aways —

- it is feasible to dispatch hundred thousand passengers, or more, per hour with a desktop computer.
- it is realizable to efficiently construct shared routes with 4+ passengers (a pool).
- a cab can serve 15+ passengers per hour on average.
- it is still reasonable to implement algorithms in C in 2023, not in garbage-collecting languages.

I. 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'), but most importantly we need to assign many passengers to one bus in such a way that passengers' time expectations are met. This leads us to a few challenges and questions:

1) classic assignment problem with huge models – one million variables and more. Many companies use least cost method (LCM, aka greedy), a suboptimal solution, but is it necessary and what is the cost? Potentially a huge one.

2) should a request be dispatched immediately with LCM or should dispatcher wait for more requests and find optimal assignments, check how they relate to each other - which passengers should share a trip and what is the optimal pickup/drop-off order? This check later called "pool finder" is another computationally demanding task, dynamic programming may give us a hand.

3) in both cases (wait or dispatch immediately) we can check if the request matches, perfectly or not, previously assigned requests. "Route extender" will do it.

4) meta optimization – suboptimal but faster algorithms may be preferable as there might be thousands of customers waiting for a trip plan.

5) how to evaluate algorithms, what is the goal function? Do we need many goal functions for different situations?

6) how does implementation technology influence performance? Is there any vital difference between current state-of-the-art frameworks?

There have been numerous studies covering optimization of taxi services, an extensive overview is given by [7]. The approach described below assumes that location of stops is known, and their number is limited. Thus, it is more a bus dispatcher than a taxi one, although taxi pick-up and drop-off places could be virtually assigned to nearby fixed stops. Number of passengers that could be assigned to one bus turned out to exceed capacity of a simple taxicab, and stopping at a bus stop can significantly increase trip sharing.

II. Assignment problem

This classic assignment problem [9] 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 kilometres 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:

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

$$\sum_{i=1}^{n} x_{ij} = 1 \; for \; j = 1, \dots, n \quad (2)$$

$$\sum_{j=1}^{n} x_{ij} = 1 \; for \; i = 1, \dots, n \quad (3)$$

$$x_{ij} \in \{0,1\} \; for \; i,j = 1, \dots, n \quad (4)$$

III. GLPK SOLVER, MUNKRES AND LCM

Three tools to solve this model have been tested – GLPK solver [10], Munkres (aka Kuhn-Munkres, Hungarian [4]) and least cost method ([8]). GLPK solver can give the flexibility to extend constraints and it helped in verification of Munkres algorithm. GLPK programme 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 Intel i7 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 I. 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 [11] and one in Rust [14].

Rust 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, although concerning another aspect of taxi scheduling, that "*optimal solutions to the offline taxi routing problem are usually significantly superior to the output of common local-improvement and greedy routing algorithms*" ([2]).

Worth noticing is that most of taxi dispatching software only uses LCM, new greedy algorithms are being developed ([3]).

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 next chapter). When requests are dispatched immediately on arrival, this is also an LCM scenario.

TABLE I. SOLVER EXECUTION TIME IN SECONDS (BALANCED MODELS)

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

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 11000 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 [12].

IV. ROUTE EXTENDER

The very first thing we can do with an incoming transportation request is to try to allocate it to an existing route (pool) if matching perfectly or non-perfectly. If it does not match perfectly (one or two stops have to 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 total trip time of added passenger (currently implemented and tested)
- minimum negative impact on the pool duration (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 IV.

V. POOL FINDER

A pool is a shared route (see also [5]) which satisfies passengers expectations – wait time and trip duration. 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 [1]. 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. After the assignment only unassigned buses and unassigned customers are given to the next dispatching phase, see below the sequence.

One assumption needs to be emphasized, both solver and pool finder need to know distances between bus stops, 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 10^{19} paths/leaves, is found after 50 seconds with Intel i7 class processor (4 threads).

TABLE II. IMPACT OF NUMBER OF REQUESTS AND MAXIMUM NUMBER OF PASSENGERS IN A POOL ON PERFORMANCE OF POOL FINDER, IN SECONDS.

n	*Three seats*	*Four seats*
100	0	2
200	1	50
300	3	-
400	9	-
500	20	-

Table II shows performance of pool finder implemented in C – three and four passengers, wait time 10min, acceptable pool loss 90% of a lone trip (detour).

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 scheduler 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, LCM)

↓

Pool finder (dynamic programming & LCM)

↓

Presolver (LCM)

↓

Solver (Munkres)

VI. 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 (see also [7], p. 21 and [6], p. 10). One can think of metrics presented in Table III. 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 at a time.

VII. 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 behaviour of all counterparts and system performance could be tested. Such environment has been implemented – simulators of customers, buses and stops (see Fig. 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 a tables IV and V 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 six rows) and scheduler's performance (last four).

TABLE III. 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 solver
average pool size (passengers per route)	The bigger the pool the fewer buses are needed
average passenger count per route's leg	Nearly a synonym of a shared route. How many seats do buses need to have?
average number of legs	How fragmented are routes? Stops costs time.
average detour	How much do customers lose on trip sharing.

TABLE IV. COMPARISON OF PERFORMANCE – WITH AND WITHOUT ROUTE EXTENDER AND POOL FINDER, 22000 REQUESTS

Metric	*No sharing*	*Ext. only*	*Pool only*	*Ext. & pool*
total buses used (assigned at least once)	7920	2173	5936	1913
max total buses assigned at a time (peak)	2580	2100	2980	1850
average wait (pick-up) time – min:secs	0:24	10:40	6:20	11:20
average trip duration– min:secs	6:21	8:35	7:15	8:55
average number of passengers per route	1	10.15	2.27	11.74
average passenger count per route's leg	0.79	1.46	0.91	1.57
average number of legs per route	1.27	16.37	3.63	19.09
average detour	0%	63%	37%	67%
max scheduler time (seconds)	0	0	12	7
average scheduler time (seconds)	0	0	4	0
average pool finder input size (demand)	-	-	111	25
average solver input size (demand)	94	3	26	7

TABLE V. COMPARISON OF PERFORMANCE – ROUTE EXTENDER VERSUS POOL FINDER, 56000 REQUESTS

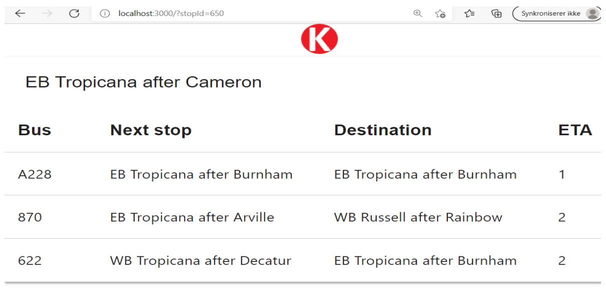
Metric	***Ext. only***	***Pool only***
total buses used (assigned at least once)	3752	9394
max total buses assigned at a time (peak)	3492	7596
average wait (pick-up) time – min:secs	11:28	6:22
average trip duration– min:secs	8:45	7:37
average number of passengers per route	14:88	2.30
average passenger count per route's leg	1.73	0.90
average number of legs per route	21.35	3.95
average detour	66%	43%
max scheduler time (seconds)	1	18
average scheduler time (seconds)	0	4
average pool finder input size (demand)	-	427
average solver input size (demand)	4	75

TABLE VI. NUMBER OF OCCUPIED SEATS IN ROUTES' LEGS – EXTENDER-ONLY SCENARIO, 22000 REQUESTS.

Number of passengers	***Number of legs***
0	5054
1	14939
2	11357
3	3359
4	802
5	198
6	37

In all scenarios requests were sent for 60 minutes, distributed evenly over time and randomly over all stops, and all had the same requirements (although the dispatcher handles individual constraints – each customer can set their own requirements) - maximum waiting time 15min, maximum detour 70%, maximum duration of trip without detour 10 min. Eleven thousand buses were available with 30km/h average speed, 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 scheduler execution time significantly, as the size of pool finder input decreased.

The most eye-catching detail is that route extender utilizes buses very efficiently. One bus transported 15 passengers per hour on average with the larger load (56k passengers), 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 [6], which was 6.


localhost:3000/?stopId=650

EB Tropicana after Cameron

Bus	Next stop	Destination	ETA
A228	EB Tropicana after Burnham	EB Tropicana after Burnham	1
870	EB Tropicana after Arville	WB Russell after Rainbow	2
622	WB Tropicana after Decatur	EB Tropicana after Burnham	2

Fig. 1. Visualization of traffic at a stop during simulation (Las Vegas)

Pool finder without extender but with more involvement of Munkres gave better wait and detour time but needed a lot more buses. The reason was the choice of goal function in extender, which was minimizing negative impact (detour) of the route being extended, not detour of the particular passenger. The same goal was chosen by [6] (p. 3).

Pool finder accompanied by route extender was the most efficient way to use buses, ten percent gain might be significant for system's profitability. In this scenario dispatcher was executed every 15 seconds in order to gather passengers, thus sending more passengers to the solver too.

With 22 thousand requests per hour the pool with four passengers (there can also be with three and to) were calculated almost in all iterations. With 56 thousand requests – almost never, it exceeded a preconfigured limit. Despite this the overall results were not worse. The reason was that there were more passengers per bus stop, the demand for buses was more congested, it was easier to find a pool, with three and two passengers. Number of passengers grew 300% but the number of needed buses only about twice.

Number of legs of a route generated by extender is significantly larger 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).

As Table VI. shows that efficient bus usage was accomplished more by proper constructing of routes than by non-random characteristics of demand and putting many passengers to one bus, occupying many seats at a time. There was no constraint during simulation concerning number of seats in a bus, but algorithm allows each bus to have its own individual constraint.

The above results were achieved on a laptop with i9-9880 mobile processor and pool finder 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 with extender only was the performance of the database running on the same hardware.

VIII. 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. Average taxi trip in New York lasts a few minutes [13]. Twenty passengers per bus per hour seems to be a restrained assessment of an average bus usage.

IX. TECHNOLOGY STACK

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 Postgres database are the only required components to run simulations, other languages are optional.

X. 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.
- driver might be needed to support disabled, inexperienced or digitally weak customers.

SUMMARY

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. 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. Result achieved during simulations are about twice as good as presented by other researchers and they were achieved under unfavourable conditions - with randomly scattered demand in Budapest (over 5000 stops) and without co-passengers (more passengers in one request sharing, family e.g.) or joining existing routes at a stop. 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] R.E. Bellman, Dynamic Programming. New Jersey, Princeton University Press, 1957.

[2] D. Bertsimas, P. Jaillet, S. Martin, "Online Vehicle Routing: The Edge of Optimization in Large-Scale Applications", Operations Research, 2019, 67. 10.1287/opre.2018.1763.

[3] Y. Kim, Y. Young, "Zone-Agnostic Greedy Taxi Dispatch Algorithm Based on Contextual Matching Matrix for Efficient Maximization of Revenue and Profit", Electronics 10, no. 21: 2653, 2021.

[4] H. W. Kuhn, "The Hungarian method for the assignment problem", Naval research logistics quarterly 2 (1955), 83- 97.

[5] P. Lalos, A. Korres, C. K. Datsikas, G. S. Tombras and K. Peppas, "A Framework for Dynamic Car and Taxi Pools with the Use of Positioning Systems," Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, Athens, Greece, 2009, pp. 385-391

[6] O.W. Ma, Y. Zheng, "Real-time city-scale taxi ridesharing," IEEE Transactions on Knowledge Discovery and Data Engineering, vol. 27, pp. 1782–1795, 2015

[7] M. Maciejewski, K. Nagel, „Simulation and Dynamic Optimization of Taxi Services in MATSim", Transp. Sci, pp.10-13, 2013

[8] H. A. Taha, Operations Research An Introduction (10th ed.). Pearson, 2017.

[9] H. M. Wagner, Principles of Operations Research with Applications to Managerial Decisions, (2nd ed.). New Jersey: Prentice – Hall, 1969.

[10] https://www.gnu.org/software/glpk/

[11] https://github.com/xg590/munkres

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

[13] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[14] https://crates.io/crates/hungarian