

Dispatching robotaxis with passenger pools – computational perspective

Bogusz Jelinski

September 13th, 2022

Abstract

The focus of making collective urban transportation more competitive and attractive is on self-driving capabilities. But do we know how to effectively dispatch thousands of cabs or minibuses, thousands of 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 & drop-off order? Maybe “best” is not the best one as we do not want passengers to wait for an optimal solution for too long, and “good” is better? Does the choice of programming tools and frameworks still play a significant role in performance? Answers have been given, a four-step dispatching process has been proposed. The spotlight is on practical aspects, not well-known theory. Implementation of algorithms (robotaxi dispatcher) and simulation environment is available as open-source on Github.

Keywords: optimal robotaxi dispatching scheduling pool cab sharing open-source

Key take-aways:

1. we can have a better urban transportation today, a taxi service on demand at a cost of a bus ticket, there is no technology enabler missing.
2. it is feasible to dispatch hundred thousand passengers, or more, per hour with a single processor.
3. it is realizable to construct shared routes with even 10 passengers (a pool).
4. Munkres (aka Kuhn-Munkres, Hungarian) algorithm solves huge models like $n=500$ in a few seconds
5. it is still reasonable to implement algorithms in C or Rust in 2022, not in C# or Java.

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 and most importantly – we need to assign many passengers to one cab 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. pool finder – another computationally demanding task, dynamic programming may give us a hand.
3. meta optimization – suboptimal but faster algorithms may be preferable as there might be thousands of customers waiting for a trip plan.
4. how to evaluate algorithms, what is the goal function? Do we need many goal functions for different situations?
5. how does implementation technology influence performance? Is there any vital difference between current state-of-the-art frameworks?

Assignment problem

This classic assignment problem [1] seems to be a challenge at first glance – the complete decision set is a permutation, which implies $n!$ size in a balanced transportation scenario, $n \times n$ variables and $2 \times n$ constraints. An academic approach would be to define a binary integer linear problem with a cost matrix and supply and demand constraints. The expected solution is a plan (cabs 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 should be constructed based on information about distances between pick-up points and current location of all cabs – see next chapter, 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:

$$\begin{aligned} \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij} &\rightarrow MIN \\ \sum_{i=1}^n x_{ij} &= 1 \quad \text{for } j=1, \dots, n \\ \sum_{j=1}^n x_{ij} &= 1 \quad \text{for } i=1, \dots, n \\ x_{ij} &\in \{0, 1\} \quad \text{for } i, j = 1, \dots, n \end{aligned}$$

Solver, Munkres and LCM

Three methods to solve this model have been tested – GLPK solver [2], Munkres and *least cost method* [4]. 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 cabs available, the time gets doubled with every hundred cabs, 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 [5] and one in Rust [9].

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

Table 1. Solver execution time in seconds (balanced scenarios)

One could ask the question why we do not use *least cost method*, “it is good enough”. No, it is not – 300 simulations with $n=200$ (cabs and passengers randomly scattered in Budapest) show that 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. Astoundingly most of taxi dispatching software only uses LCM. We still need LCM in two cases – models too big to be sent to Munkres that need to be squeezed by LCM “presolver”, and we need LCM in pool finder. The latter and route extender described below are two steps prior to solver stage, which can significantly reduce the size of the

assignment model. LCM presolver might not be involved with load below 2000 requests per minute. All these four steps have been implemented in Kabina dispatcher [6].

Pool finder

A pool is a shared route which satisfies passengers expectations – wait time and trip duration. A trip 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 → 2out

An effective way to *search* through such trees is dynamic programming [7]. In short you do not recurse over nodes all over again, but you divide search in stages starting from the leaves. For example you check which of the two is better - “3 out → 1 out → 2 out” or “3 out → 2 out → 1 out”, and when you go level up and add the forth passenger (“4 out”) you do not need to care about “1 out” and “2 out”, which saves time in the next stage, their constraints have been checked and result stored in memory. Here comes an interesting tuning issue – it takes time to remove such duplicates (1out → 2out vs 2out → 1out), which can come from different computing threads. It turns out it is faster not to do it in pool finder. Important part of reducing the size of memory used by individual stages, reducing number of branches stored at them, is the check of wait time and trip duration at each stage. The more restrictive passengers are, e.g. the smaller the loss of time in a pool (detour) they want to accept, the faster is the pool finder.

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. It might be it, the first “in” passenger would be assigned a cab by the solver, but there is a catch. This assignment must meet requirements of all pool participants, not just the first one in pool’s sequence, we risk dropping a lot of shared trips. The assignment must be done in the “pool finder” method, in the loop where repeats are removed, LCM is a natural choice. After the assignment the assigned cab is not sent 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 with real data from the field.

The gain of using dynamic programming is significant – a solution for 200 requests with 4 sits (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 2 shows performance of pool finder implemented in C – three and four passengers, wait time 10min, acceptable pool loss 90% of trip without pool (detour).

n	3 sits	4 sits
100	0	2
200	1	50
300	3	-
400	9	-
500	20	-

Table 2. Performance of pool finder implemented in C, in seconds.
Impact of number of requests and maximum number of passengers in a pool.

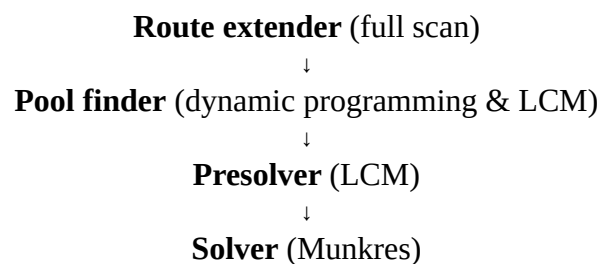
If executed several times per minute this pool finder followed by three-step solver stack (responsible for assignment of trip requests which cannot be shared in a pool) can assign at least one hundred thousand requests per hour with a low-end processor with four cores. Today's high-end chips host several dozen cores.

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, so we must choose what we are interested in (goal function), for example:

- minimum total trip time of added passenger (currently implemented in Kabina)
- minimum negative impact on the pool duration
- minimum total detour of all passengers

With this step the total dispatching process looks like this (added comment on *operations research* techniques used):



Even more passengers

There are some techniques and functionalities that may help put more passengers into a cab. Most importantly these techniques are not computationally heavy:

- allowing for taking co-passengers - one trip request with more than one passenger
- allowing for joining an already defined route at the bus stop; passengers 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 chose 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 transport several hundred thousand passengers per hour. How many buses would be needed depends on the allowed travel time. Average taxi trip in New York lasts a few minutes [8]. Twenty passengers per bus per hour seems to be a restrained assessment of an average bus usage.

Quality of routes

Simple time constraints can lead to strange routes. A cab 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 is to promote more straight routes with stop bearings. Kabina uses this approach including a route visualization tool thought of as route quality check, see Figure 1.

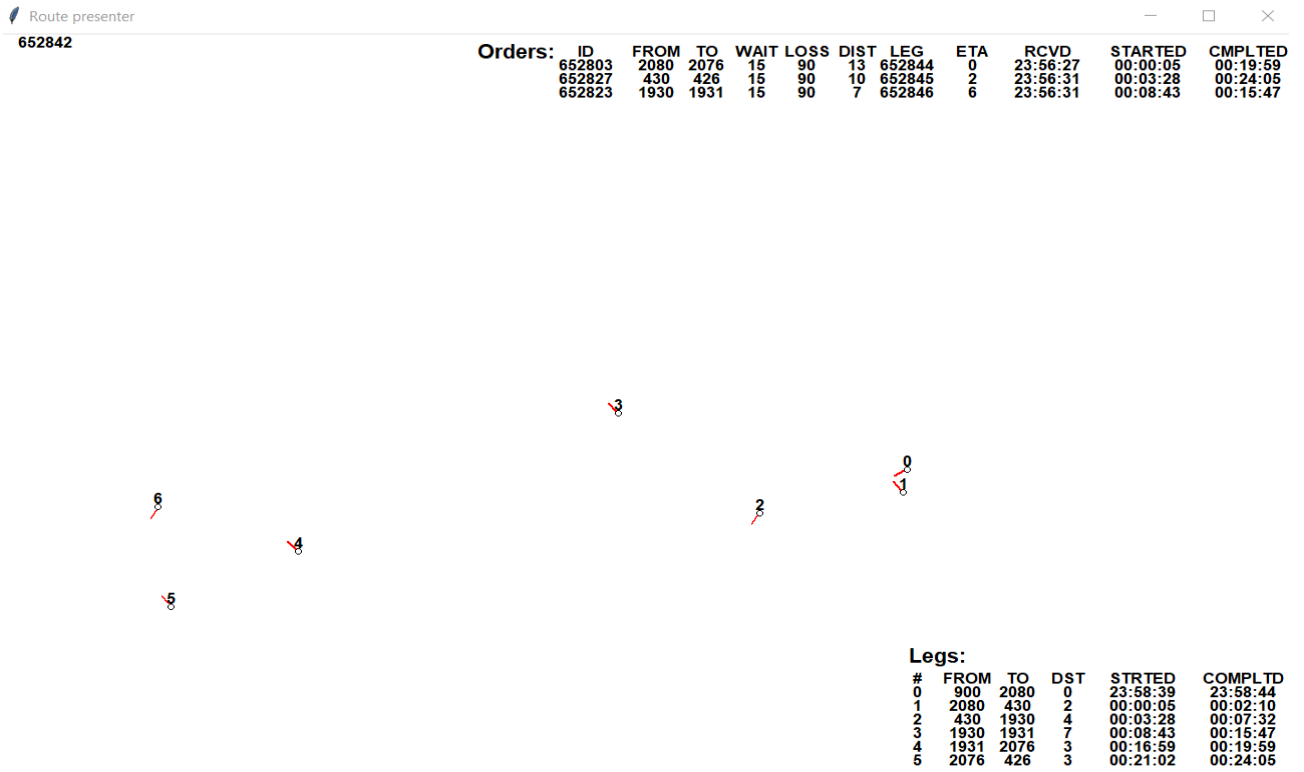


Figure 1. Kabina route validator

Simulator

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, APIs and service flows, not least charging models that could impact total revenue. Is it better to run the dispatcher more often? Smaller data sets could mean worse plans but faster response. Therefore, it is desirable to have a simulation environment where behaviour of all actors and system performance could be tested. Kabina comes with a proposal of such environment – simulators of customers, cabs and stops (see Figure 2), supplemented by RestAPI and database storage. We could measure some metrics to compare dispatchers, have a look below how route extender have affected results:

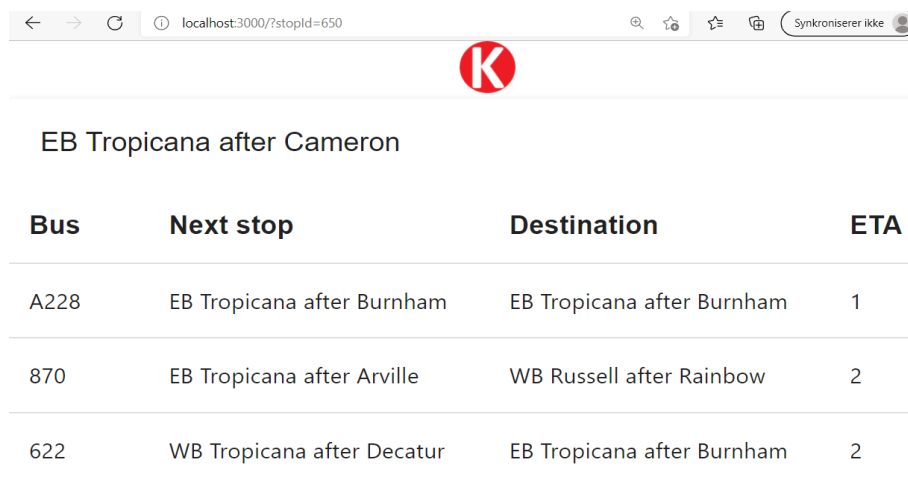
Metric	Without route extender	With route extender
total wait (pick-up) time – hours:min	2454:31	2851:17
total duration of all requests	3762:05	3743:07
total drive time of cabs	2462:26	2152:15
average pool size (passengers per route)	2.18	2.80
average passenger count per route leg	1.03	1.14
total number of customers rejected	0	88
total cabs used	3918	3793
total cabs assigned at a time	3398	3163
average scheduler time (seconds)	15	10
max scheduler time (seconds)	47	49
average pool input size	155	130

Table 3. Comparison of performance – without and with route extender

In both situations 23171 orders were submitted within 55 minutes, which is above 25 thousand per hour (max wait 15min, max 70% detour, max trip 10min), 4000 buses were available (60km/h average speed¹), routes were limited to 10 stops. Interestingly, additional logic of the extender (and its execution time) decreased total scheduler execution time significantly, as the size of pool finder input decreased. While travel time increased, cab utilization improved slightly - without extender many customers would have been assigned a dedicated bus. The biggest advantage is decreased risk of exceeding the limits of the pool finder. Limits are customizable depending on the hardware performance, limits are intended to protect against requests piling up while the scheduler is working. As Table 4. shows quite many could be able to share a trip.

Number of Orders (passengers)	Number of routes (without route extender)	Number of routes (with extender)
1	3650	1533
2	1971	1033
3	4277	3474
4	687	1911
5	0	283
6	0	14
7	0	1

Table 4. Orders count in routes (with route extender and without)



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

Figure 2. Visualization of traffic at a stop during simulation (Las Vegas)

Technology stack

At the beginning all in Kabina was written in Java and TypeScript (frontend applications). Pool finder is now also implemented in Rust, C# and C. Java and C#, including multithreading, are at least 3 times slower than C while tested on Windows 11. 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

¹ That might seem unrealistic, you need more cabs if they move slower.

simulating customers and cabs have been rewritten in Golang. Java is not the fastest RestAPI framework either, Rust is used now.

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 for 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
- driver's presence will improve passengers' comfort and security
- driver might be needed to support disabled, inexperienced or digitally weak customers.

Summary

Due to lack of infrastructure, it was not possible to run full scale (100k+ requests/h) simulations with client simulators, RestAPI and dispatcher. One hundred thousand trips requests means about two million http requests to API per hour, but this part can be linearly scaled. Small simulations (25000 passengers/h run on 8th generation Intel i9) turned out to have **three passengers in pools on average**, less than 10% drove alone. Average assignment time was 15sec, average wait time for a cab was 6min, average detour was 49%², a cab carried 9 passengers per hour on average. This result was achieved 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 we know that passengers' streams are more concentrated around some main routes and transport hubs, especially during rush hours, which means we can achieve better results in real life.

References

- [1] <http://web.mit.edu/15.053/www/AMP-Chapter-09.pdf>
- [2] <https://www.gnu.org/software/glpk/>
- [3] https://en.wikipedia.org/wiki/Hungarian_algorithm
- [4] <https://cbom.atozmath.com/example/CBOM/Transportation.aspx?he=e&q=lcm>
- [5] <https://github.com/xg590/munkres>
- [6] <https://github.com/boguszjelinski/kern>
- [7] https://en.wikipedia.org/wiki/Dynamic_programming
- [8] <https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page>
- [9] <https://crates.io/crates/hungarian>

2 (average trip length – average requested distance)/average requested distance = (9.74-6.55)/6.55