Optimal taxi dispatching with a thousand cabs
Tooling for future self-driving taxi services

Bogusz Jelinski
[DRAFT]
October 7th, 2020

Abstract

Serving transportation requests coming from taxi customers can be modeled (and solved) as a binary integer transportation problem with sources, destinations and total wait time to be minimized. There are plenty of scientific papers covering this topic ([4]). Although the basics are presented below I am concentrating on practical aspects – a solver widely used to deal with such models, “pools” and how to solve problems that are too big for a solver. The interesting part in it is that in real-world situations an assignment model can have millions of variables – number of customers * number of cabs. A solution for such a scenario is discussed. While combining a solver with a pool and greedy (lowest cost method) pre-solver, most challenging scenarios like the demand in New York can be served and it has been demonstrated in form of a simple simulator, which served tens of thousands of customers per hour on a single processor thread. All source code and output of simulations are available in GitHub[1] for reproducibility. A solution for pools with four passengers is included.

Keywords: optimal taxi dispatching pool Java Python Julia

1 Introduction

We will tackle with taxi scheduling problem – how to respond to demand of customers (potential passengers) by assigning cabs to requested trips from point A to B. This classic assignment problem [1] seems to be a challenge at first glance – the complete decision set is a permutation, which implies a *n! s*ize in a balanced transportation scenario. A common approach would be to define a binary integer linear problem with supply and demand constraints. The expected solution is a plan – a cub assigned to a request which 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 but less profitable if customers get charged per distance.
In a balanced problem (number of requested trips equals number of cabs) we would have a cost matrix like the one shown in table 1. The 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 will take. In advanced scenarios average time can depend on the time of day, it can vary due to rush hours.

Passenger Cab	1	2	...	n
1	c_{11}	c_{12}	...	c_{1n}
2	c_{21}	c_{22}	...	c_{2n}
...	...	...	...	$c_{?n}$
n	c_{n1}	c_{n2}	$c_{n?}$	c_{nn}

Table 1: Balanced transportation problem - cost matrix

That means n*n variables and 2*n constraints. The cost matrix is not likely to be symmetric in real life – some streets can be one way. Formally it has the following form:

1 https://github.com/boguszjelinski/taxidispatcher

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

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \text{for j=1,...,n}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \text{for i=1,...,n}$$

$$x_{ij} \in \{0,1\} \quad \text{for i,j = 1,...,n}$$

In a non-balanced scenario the first or the second set of constraints will be inequalities, ≤ 1. With other words – some cabs will not be assigned a trip (supply > demand) or a customer will have to wait for a free cab (supply < demand). Another option is to add dummy cabs or dummy customers to balance their number, with costs, say, ten times bigger than highest cost in the matrix – see code examples in appendices[2].

2 Implementation

Let us begin with a simple example and assume there are six pick-up / drop-off points (taxi stands) in a city, four trips have been requested between these stands and there are three cars in service at a particular point in time. To construct the cost matrix we need three input matrices – distances between taxi stands, demand requested and current location of cubs. While minimizing pick-up wait time we would like to have information about current tasks performed – maybe a cab completes its trip nearby soon? On the other hand – in rush hours demand can change a lot in a matter of minutes and response plan should be adjusted. The example will be simplified, only the information about end stand will be used, we assume de facto that the cabs are already there and stand still. In a more usable approach the information about start time could be used, which could give us estimated time of completion of the current trip.

To stand From stand	1	2	3	4	5	6
1	0	1	3	4	7	9
2	1	0	2	3	6	8
3	2	3	0	1	4	6
4	5	3	1	0	3	5
5	7	6	4	3	0	2
6	9	9	5	5	1	0

Table 2: Distances between stands in km

To stand From stand	1	2	3	4	5	6
1			1			2
2						
3						
4		3				
5						
6		4				

Table 3: New demand – new trips requested (numbers in matrix denote passenger indices)

2 See *python.py* on GitHub [3]

To stand / From stand	1	2	3	4	5	6
1						3
2						
3						
4		2		1		
5						
6						

Table 4: Current demand – trips in progress (numbers in matrix denote car indices)

To sum it up – cub1 is not moving and waiting at stand4, cub2 is heading stand2 and cub3 will stop at stand6. We have to customers at stand1 heading stand3 and stand6 and to other customers are heading stand2 from stand4 and stand6.
Then the cost matrix in our linear model will be as shown in table 5 - sum of distance to pick-up a customer and the requested trip itself.

Customer / Cab	1	2	3	4
1	5	5	0	5
2	1	1	3	8
3	9	9	5	0

Table 5: Cost matrix

The first cell (cab1 assigned to customer1) is ’5’ as cab1 has to move from stand4 to stand1 and take the customer to stand3 (see the costs in table 2).
We will have the following constraints:

$$\sum_{i=1}^{3} x_{ij} \leqslant 1 \quad \text{for j=1,...,4}$$

$$\sum_{j=1}^{4} x_{ij} = 1 \quad \text{for i=1,...,3}$$

That means sum of one of four columns will be zero, one of customers will not get a taxi.

In appendices you will find implementation of the model in Python and Julia. In appendix 3[3] you will find the code used for performance tests – a plan for 1000 cabs serving 1000 passengers is found below two minutes on i7-4790 processor. Tests with Python gave slower executions maybe due to difference in GLPK versions. Keep in mind that GLPK used here is not the fastest solver[4] and you can get a faster processor today – with at least 50% faster single thread performance. Additionally «pool» described below can significantly reduce number of variables in the model. Number of stands where cabs can stop (from which passengers can begin their trip) can be much larger, it does not affect the performance. Dividing a bigger problem into smaller ones based on e.g. districts delivers sub-optimal solutions – see chapter 5.

This model is far from a real-life implementation but its goal was to show that solving huge linear models is feasible. Let us have a look at assumptions and some simplifications:

3 See *perf.jl* on GitHub [3]
4 check cplex, Gurobi, scip, or CBC

- start and end of a trip is a taxi stand,
- customers request cabs ASAP, but in reality most of them can be ’at a particular time’, ‘in 5 minutes time’ for example.
- some cabs are about to run out of battery or drivers will take a break – they will not be available
- distances (travel time) can vary throughout the day – mean values for particular hours will be probably collected by taxi operators
- passengers do have a preference how long they can wait before they chose some other option – depends on how well public transport is developed in their regions
- some customers are prioritized (VIPs), they should have equality in their constraint in the model.

3 Verification

In order to make sure that solver gives good results and does not fail one thousand checks (random generated problems) were run and the objective value was compared to Lowest Cost Method (LCM) [2] – see Fig. 1. This was calculated with n=100 – one hundred cabs and the same number of customers[5].

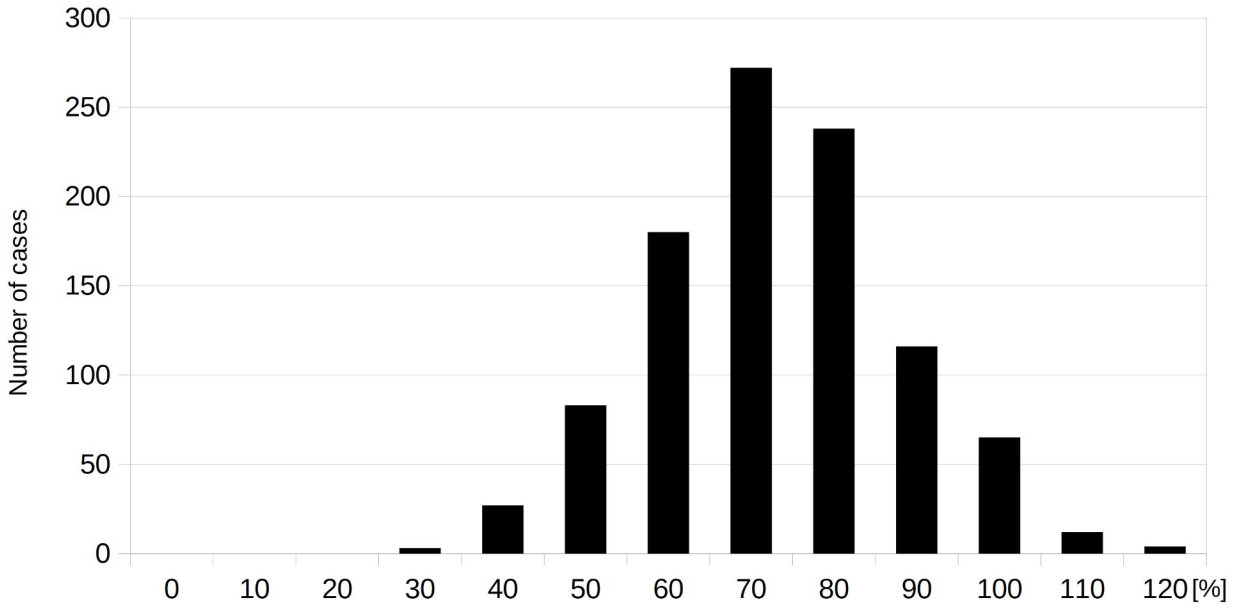

Figure 1. How much worse are results of LCM [%]

This could suggest that huge savings could be achieved while using solvers – both fuel and customers’ wait time can be saved. LCM value was on average 78% higher than the optimal one. But this result was achieved with a random generated cost matrix – as shown later with non-random distances LCM can significantly reduce time of computation without affecting results to this extent. To illustrate why LCM does not give the optimal solution a simple example was given below with two customers and two cars – see Fig. 2.

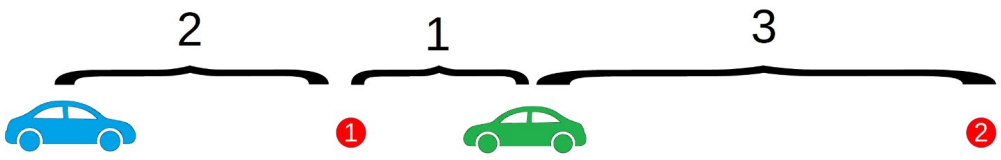

Figure 2. Example to compare LCM against the optimal solution

5 See *heuristic.py* on GitHub [3]

The LCM would look for the nearest customers (shortest distances) thus matching the green car with the customer ‘1’ first, then it would match the blue car to customer ‘2’. The total distance to drive would be 1 + 6, if cars and customers are lying on a line. But the optimal solution is to match the green car to customer ‘2’ and the blue one to ‘1’ resulting in a 3+2 plan.

4 Pool

Sharing a trip with another passenger is a step forward in making human transportation more sustainable. UberPool is an example of this idea. The problem now will be to choose passengers that should be proposed a shared trip – not all of them will agree to such a trip and even if agreed most of them will most likely have a preference of how much their trip can be extended, and not longer. Let us assume it can be only two passengers in a cab, although a passenger means a party ordering a trip – a whole family too. A passenger that is picked up by another passenger can be expelled from demand in the model described earlier, which decreases its computational complexity. Now we can minimize the trip itself (its distance), not the distance to pick up a passenger. Constraints are different - if passenger1 picks up passenger2 then passenger4 cannot pick up passenger1:

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

$$\sum_{i=1}^{n} x_{ij} + \sum_{i=1}^{n} x_{ji} = 1 \quad \text{for j=1,...,n}$$

$$x_{ij} \in \{0,1\} \quad \text{for i,j = 1,...,n}$$

That makes constraints strong enough to make GLPK fail quite often[6]. The good news is that based on some dozens checks (code included in GitHub) the LCM method for this scenario gives results not worse than 3%, 2% on average when compared to optimal ones[7], and completes after a few seconds for n=1000 if written in C[8] and as shown later on during a simulation – a Java implementation is quick enough for even the most challenging scenarios. It is challenging as it is a *n^p* task[9], where ‘p’ is the number of passengers that fit into the cab. That means that theoretically there is one trillion checks to be made when trying to put 1000 passengers in four seats.
Such a LCM program consists of three steps – calculating costs of all combinations of two passengers, sorting the result and elimination of pairs were one of its passengers appears in a less costly combination. As demonstrated by the Pool.java example when additional checks are applied (maximum wait time for a cab) even four passengers can be assigned within seconds.

5 Split of computation

If the number of requests from customers makes the computation last longer than one can accept, the task can be divided into smaller subtasks computed separately and simultaneously. The overall result will be suboptimal but this solution is still better than LCM on average - a check with 1000 cases of size n=400 divided in four smaller tasks proved to be 16% worse on average than optimal, see Fig. 3. In fact there were five tasks, as customers not served by these four main ones were allocated by the fifth, final task. LCM proved to be 20% worse but when better result is picked up from the two (sometimes LCM gives a better result) – the result decreases to 14%.

6 See *pool_optimum.py* on GitHub [3]
7 See *pool_opt_min.py*
8 See *pool.c*
9 Nearly. It is a variation without repetition.

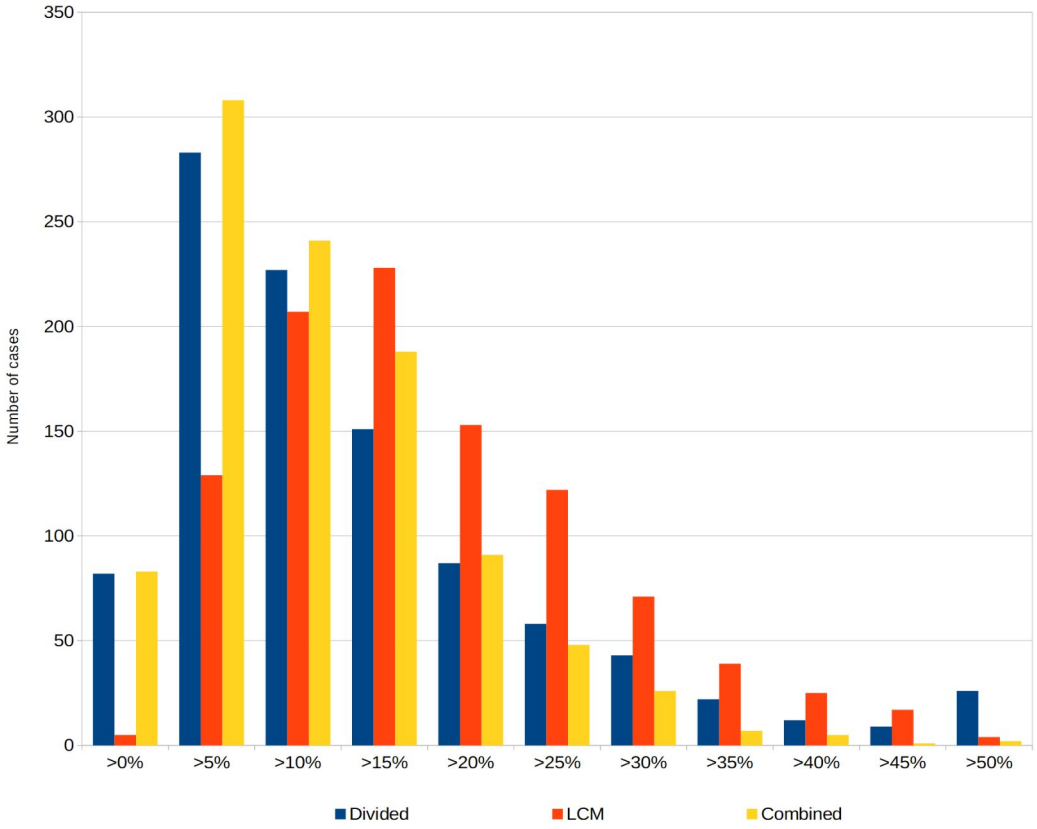

Figure 3. Degradation of results by dividing computation into four tasks [%]

Even better results were achieved when using LCM as a pre-solver for shortest distances and then looking for optimal solution with the rest of customers and cabs not assigned by LCM[10] – results degraded by only 2.4% while the size of model sent to GLPK was reduced from n=400 to 146 on average. It was presented on Fig. 3 as “combined”.

6 Simulation

To show how it all can work in a real-life solution a simulator was written[11] - a simplistic one in Python and one in Java with a customer pool, although the latter one calls Python solver module – Java GLPK interface is not efficient with such number of variables. Random demand from customers comes at each iteration (let us assume – an iteration is a unit of time), cabs get assigned and move to pickup customers and take them to their destinations, pool customers are assigned to these that are going to pick them up – only the latter ones are sent to the solver. The most challenging scenarios like that from New York City (about 30000 customers per hour with a few miles/minutes of an average trip) are achievable for the methods discussed here – the size of a model sent to the GLPK solver should not exceed n=1000, but with a pool and LCM preprocessor n=4000 are within reach of an average home computer.

7 Summary

The days where self-driving cabs will be a common-place in our cities is not a distant future. Its operators will need efficient algorithms to provide both competitive user experience and profitability while running on low margins. That is why finding optimal dispatch plans is so important. As shown above it is possible to make better plans that these based only on LCM, a solver for tasks that suites today’s average cities (or city districts in big cities) is at hand, plans with four passengers in a cab can be found with the computing power available today.

10 See *greedy_opy.py*
11 See *simulator.py*, *gendemand.py* (a generator of simulation input), *Simulator.java* and *solver.py*

Appendix 1 – Implementation in Python

```
from cvxopt.glpk import ilp
import numpy as np
from cvxopt import matrix
n=4
c=matrix( [5,5,0,5, 1,1,3,8, 9,9,5,0, 100,100,100,100], tc='d')
a=matrix([ [1,1,1,1, 0,0,0,0, 0,0,0,0, 0,0,0,0],
       [0,0,0,0, 1,1,1,1, 0,0,0,0, 0,0,0,0],
       [0,0,0,0, 0,0,0,0, 1,1,1,1, 0,0,0,0],
       [0,0,0,0, 0,0,0,0, 0,0,0,0, 1,1,1,1],
       [1,0,0,0, 1,0,0,0, 1,0,0,0, 1,0,0,0],
       [0,1,0,0, 0,1,0,0, 0,1,0,0, 0,1,0,0],
       [0,0,1,0, 0,0,1,0, 0,0,1,0, 0,0,1,0],
       [0,0,0,1, 0,0,0,1, 0,0,0,1, 0,0,0,1]
       ],tc='d')
g=matrix([ [0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0] ], tc='d')
b=matrix(1*np.ones(n*2))
h=matrix(0*np.ones(1))
I=set(range(n*n))
B=set(range(n*n))
(status,x)=ilp(c,g.T,h,a.T,b,I,B)
status
print(x)
```

Appendix 2 - Implementation in Julia

```
using JuMP, GLPK
model =  Model(with_optimizer(GLPK.Optimizer))
n=3
m=4
t = [5 5 0 5; 1 1 3 8; 9 9 5 0]
@variable(model, x[1:n,1:m], Bin);
@objective(model, Min, sum(t[i,j]*x[i,j] for i in 1:n, j in 1:m));
@constraint(model, [i=1:n], sum(x[i,j] for j in 1:m) == 1);
@constraint(model, [j=1:m], sum(x[i,j] for i in 1:n) <= 1);
optimize!(model)
for i= 1:n
   for j= 1:m
      println(value(x[i,j]))
   end
end
```

Appendix 3 – Code used in performance tests

```
using JuMP, GLPK
model =  Model(with_optimizer(GLPK.Optimizer))
n=1000
m=n
t = rand(10:40,n,n)
@variable(model, x[1:n,1:m], Bin);
@objective(model, Min, sum(t[i,j]*x[i,j] for i in 1:n, j in 1:m));
@constraint(model, [i=1:n], sum(x[i,j] for j in 1:m) == 1);
@constraint(model, [j=1:m], sum(x[i,j] for i in 1:n) == 1);
optimize!(model)
```

References

[1] http://web.mit.edu/15.053/www/AMP-Chapter-09.pdf

[2] https://www.thinkmind.org/download.php?articleid=iccgi_2015_6_30_10116

[3] https://github.com/boguszjelinski/taxidispatcher

[4] http://www.mit.edu/~jaillet/general/online-routing-18.pdf

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