Optimal taxi dispatching with a thousand cabs
Tooling for integer linear approach with a million variables

Bogusz Jelinski
August 10th, 2019

Abstract

Serving transportation requests coming from taxi passengers can be modeled 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. In this article I am concentrating on practical aspects - solvers widely used to deal with such models and implementation hints. The interesting part in it is that in real-world situations an assignment model can have millions of variables making some people think of implementing heuristics, which does not seem necessary. A solver for such a challenging scenario is discussed.

Keywords: optimal taxi dispatching Python Julia pool

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!* size 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.
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.

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:

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

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

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

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

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 Implementation

Let us begin with a simple example and assume there are six pick-up points (taxi stands) in a city, four trips have been requested 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 – the first example will be simplified, only the information about end stand will be used. In a more usable approach later on I will use information about start time, 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)

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 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 (check Gurobi!) 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.

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:
- 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’
- 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 insure 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.

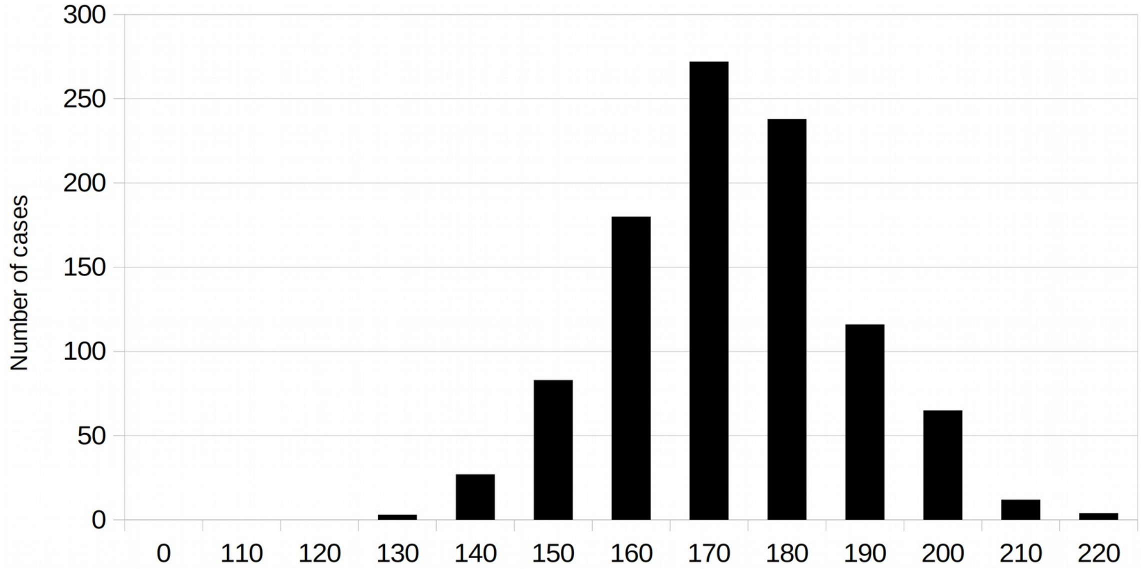

Figure 1. Comparison of optimal objective value with LCM – LCM/optimal*100

This shows how huge savings can 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.

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 as if passenger1 pick up passenger2 then passenger4 cannot pickup 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. 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, and completes after a few seconds for n=1000 if written in C, see [3] to get the code. Such a 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 le
ss costly combination.

6 Summary

The days where self-driving taxi 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 schedules is so important. As shown above solvers for model sizes that suite today's average cities (or city districts in big cities) are available and they can give significant, double-digit savings.

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