Finding passenger pools with dynamic programming

How to beat Uber by allocating 10+ passengers to one minibus route

Bogusz Jelinski, March 24th, 2021

Keywords: taxi minibus pool dispatcher GLPK LCM Uber

The temptation

Public transport in cities would be totally disrupted if we could offer a cab trip at the cost of a bus ticket. A trip that offers a short pickup time and takes you directly to your destination. Then we need a solution which is much better than the one offered by Uber in terms of cost efficiency. We need buses with routes created on request with much more passengers to split the cost of the driver, among other factors. Yes, legislative acceptance for self-driving might not come that fast, speed limit might not be acceptable (in most city projects today it is 25km/h, accidents will happen), we might still need someone on board because of possible fraud, vandalism, security issues, convenience / comfort, maintenance or to deal with unpredictable unknowns which could ruin user experience.

One of the most challenging computational problems in this task is to gather together trip requests which share a direction, maybe only partially. Such trip is often called a pool. The order with which to pick up passengers will in most cases be different from the drop-off plan, and there might be many feasible plans. With other words – it will be a permutation. A full scan for 500 requests with four passengers in pool gives nearly 500^8 checks – nearly 4 sextillions in short scale. Although with some assumptions and a hundred processor cores it could still be possible to run a full scan, there is a more intelligent solution – dynamic programming, dividing the permutation tree in stages and finding optimal solutions for underlying subproblems, starting with leaves.

The solution

Dynamic programming does have its own challenges – extensive memory usage, so a balance needs to be found. The proposal here is to divide the whole tree into two subtrees – pickup and drop-off, and merge the trees with a hash map, a list of sorted passengers being the hash map key. We shall start with leaves for each tree, which are pairs of last drop-offs:

1) we check if the last passenger is happy with its predecessor – if the distance is not longer than a direct trip plus acceptable loss. Then we store acceptable plans, e.g. 1-2 (we have checked that ’2’ is happy) and 2-1 (‘1’ is happy too, we have checked this)

2) at the next stage we iterate over all passengers and check if a plan is acceptable, e.g. 3-1-2 and 3-2-1. Here we again must check if 1 and 2 are still happy, but we can remove the leaf with longer distance. All these checks takes time but it cuts a lot of branches, the reward is comming. We cannot check if ‘3’ is happy with the plan as we will know its predecessor in the next stage. And so on ...

3) at the root of the drop-off tree we save all acceptable plans in a hash map with a key which is a sorted list of passengers in the tree. A plan from the pick-up tree will also come with such a key. When the key is found in the drop-off tree we must check again if all passengers are happy.

4) all acceptable plans are sorted by total distance, duplicates with longer distances get removed.

A few figures – with 300 requests and 50% acceptable loss both trees have about 5k leaves (theoretically there could be nearly 300^2), 250k entities in root stages, 3k entities after merge (great!), about 30 pools, which means 30*4 passengers assigned to pools. With 400 requests the heap size in Java must be extended to several gigabytes. It is obvious that after running the

algorithm for four passengers one should run it for three and two, as some passengers' requests suite only smaller pools. There are some model parameters which affect these figures – number of stops and maximum trip duration.

There are two assumptions – we know distances between bus stops and requests coming from passengers contain their preferences, how much they can lose while sharing a ride, are they in a hurry and need a dedicated cab or they prefer a slower but cheaper trips. This preference helps cut off unacceptable tree branches early, thus reducing memory needed and boosting performance.

The difference in performance between full scan and dynamic programming is of two orders of magnitude – it takes a Java version of full scan 160 seconds to service 200 requests with 50% loss and put them into pools of four. The "dynamic" version takes **under one second**. 100% loss requires two seconds as it makes more permutations acceptable. These results come from an Intel i7 home computer. With implementation in C (work in progress) and executed on a high-end hardware, pools with five passengers could be computed.
There is only one way to understand how the algorithm works in detail – by analyzing the source code provided, see the open-source **Kabina** project, Kaboot repository here:
https://gitlab.com/kabina/kaboot
The good news is that its core is around 100 lines.

The road from 5 to 10+ passengers
There are many techniques that may help put more passengers into a bus than just these found by the "dynamic" stuff described above, some techniques already implemented in Kabina. Most importantly these techniques are not computationally heavy:
- allocating new requests to existing routes (pools) if perfectly matching,
- finding identical requests (identical from-to) and treating it as a one, collective request/passenger
- allowing for taking co-passengers (similar to the one above)
- allowing for joining an already defined route on the stops; 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 absolutely imaginable that combination of these techniques 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, see references. Forty passengers per bus per hour seems to be a restrained assessment.
BTW – the only limit with my home computer is the performance of RestAPI interface, the core including PostgresQL database was able to dispatch thousands of passengers per minute. SpringBoot RestAPI interfaces scale fantastically.

Other solved challenges
You have to assign buses to discovered pools or lone passengers. This classic assignment problem seems to be a challenge at first glance, such a model can have millions of variables – number of customers * number of cabs. The size of the decision set is larger than … estimated number of atoms in the universe! There is another good news today – GLPK solver (open-source) tackles such models, the Kaboot server generates GLPK programs and executes them. For bigger problems than our hardware is capable of, Kaboot offers a "greedy" pre-solver, called also low-cost method (LCM). Such a sub-optimal solution costs a few percent of the total distance (buses have to move towards customers) but can be of critical importance – customers can receive their assignments much faster.

References
https://gitlab.com/kabina/kaboot
https://github.com/boguszjelinski/taxidispatcher
https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
https://www.thinkmind.org/download.php?articleid=iccgi_2015_6_30_10116