**Optimization Methods Project Report**

Taxi Company Car Allocation

**Table of Contents**

[1. Team members & individual contributions 2](#_Toc469953678)

[2. Problem background 2](#_Toc469953679)

[3. Problem formulation 2](#_Toc469953680)

[4. Chosen methods 3](#_Toc469953681)

[5. Data 10](#_Toc469953682)

[6. Results 10](#_Toc469953683)

[7. Conclusion 11](#_Toc469953684)

[8. References 12](#_Toc469953685)

1. Team members & individual contributions

Artem Korenev – Mathematical formulation of a column generation and dynamic programming algorithms, column generation implementation, ESPPRC algorithm implementation;

Andrei Kvasov – Mathematical formulation of problem in general and research on dynamic programming approaches, dynamic programming implementation, data management and transformation, column generation research;

Anton Marin – Mathematical formulation of greedy algorithms, implementation of greedy local search and simulated annealing algorithms, testing of the dynamic programming algorithm implementation, column generation research;

Oleg Sudakov – General formal mathematical formulation of the problem, column generation research, ESPPRC algorithm research, data management and visualization implementation.

1. Problem background

It is not uncommon for taxi cars to have a different geographical distribution compared to clients. This leads to underperformance of taxi companies, as clients must spend more time, waiting for the ordered car to arrive, and cars spend more fuel during the day. An effective taxi car geographical allocation method could reduce both metrics, effectively increasing taxi company’s rate of return. Finding such method will be our aim in this project.

1. Problem formulation

The problem of finding effective taxi car geographical allocation for a given day could be split into two parts:

1. Predicting the approximate demand (times and locations of the orders) for a given day;
2. Calculating effective geographical allocation for found demand.

In our project, we are going to consider only the second part of such problem: given demand information (times, pickup and destination locations) of the orders, the effective geographical allocation will be found.

The total number of taxi drivers will be denoted as , and the number of clients will be denoted as . Every client during the day must be assigned to some taxi car. We will assume, that each taxi driver’s work shift has a following schedule:

1. Each taxi starts at a predefined location , corresponding to the depot;
2. Taxi drives to pickup location and waits for assigned client, if needed;
3. Taxi delivers the client to his/her destination point;
4. If all clients, assigned to car, are delivered, taxi heads to depot, otherwise go to step 2.

As our goal is to maximally reduce operational expenses, we will consider three factors, contributing to them:

* Client waiting time;
* Fuel consumption.
* Taxi downtime (time spent in the pickup point waiting for customer).

Therefore, for a given formulation of the problem, our goal is to:

* - linear fuel and car downtime penalty for driver and client ;
* - quadratic penalty for client ’s waiting time.

It is worth noting that the problem formulation, which includes penalty for car downtime is essentially equivalent to the problem, in which we impose a penalty on usage of each car. This penalty can be thought of as downtime cost per unit of time multiplied by driver’s work shift. Replacing car downtime penalty by a fixed one will impose a penalty even on time periods, when the driver is en route, but it won’t impact the solution, as the fuel penalty will outweigh it anyway.

By introducing penalty constants,

* – fuel cost per unit of distance;
* – downtime cost per unit of time;
* – client waiting cost per unit of time squared.

the target function components can be computed as:

* – distance from current taxi position to pickup position of client ;
* – distance from client pickup point to client destination point;
* – taxi arrival time to client pickup point;
* – client order time.

1. Chosen methods
   1. Greedy algorithm

The greedy algorithm was implemented in the following way: considering table with C rows, where each row represents some current state of the driver,

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ID | Xpos | Ypos | Finish time | Path | Cost | Assigned |
| 234 | 40.5 | -71.0 | 158 | [2,5,12] | 1575 | 1 |

Where:

* ID represents the number of the driver;
* Xpos, Ypos represent coordinates of the current driver position;
* Finish time represents time in minutes from the beginning of the day, when driver completes all previous jobs;
* Path represents array of clients, that driver has delivered until now;
* Cost represents the sum of fuel costs and clients waiting penalties, associated with this driver for his previous jobs until now;
* Assigned represents a Boolean variable, whether the driver has completed at least one job from the beginning of the day until now.

First, we initialize the table for all drivers:

* Xpos, Ypos – coordinates of the depot;
* Finish time = 0 (indicates the day start);
* Path = [] (no clients delivered from the beginning of the day);
* Cost = 0;
* Assigned = 0.

Given a sorted array of orders, sorted by pickup times, we pick next client, calculate total costs among all drivers prior delivering this client, then for each driver we calculate cost, if this driver will deliver the client and assign client to the driver, whose accumulated cost is the lowest. During calculation of the cost, we consider previous driver location (Xpos, Ypos), client position, distance for driver needed to approach client, using given speed and time. When the driver becomes free of all previous jobs, we calculate client waiting penalty and amount of burned fuel both for approaching client and its delivery. Then we update the corresponding row in the table of driver’s current state. After all clients were assigned, we add additional cost of return to depot for all drivers. Total cost of the solution is sum of costs among all drivers.

The **complexity** of algorithm is .

* 1. Local search

For local search, we define neighborhood as follows: let the table of current drivers’ states be complete, and drivers are returned to the depot. We pick random assigned driver, pick his random client from the path (copy and remove it from the path). Then we pick any another random driver (can be equal to the first driver) and insert the copied client to random position in second driver's path. Then we reevaluate costs for mentioned two drivers, as this transformation doesn't affect other drivers. By applying this transformation multiple times, we obtain any feasible solution, so our neighborhood function is defined correctly.

The **complexity** of one transformation in the worst case is .

In local search we initialize our solution by random (or greedy) one. Then we get a solution from the neighborhood and calculate its objective. If the objective of a neighbor is smaller, we go to this solution and start next iteration from it.

* 1. Simulated annealing

We initialize our solution by the greedy one, then we go to solution with better objective always, and the probability of jumping to the solution with worse objective depends on temperature, which decreases as algorithm proceeds.

* 1. Dynamic programming

For dynamic programming implementation, an additional data structure must be created: two-dimensional table, or array, of size . The table filling method is common for dynamic programming in general:

1. All clients are sorted by their pickup times;
2. We examine the first unassigned client. The optimal total accumulated cost of his assignment to driver is calculated, as in 4.1, and is written to a corresponding cell of the table. During calculation, all assignments of previously examined clients are considered.
3. After filling the column, corresponding to the last client, the minimal value in it is found, and optimal assignment is given by a backtracking procedure.

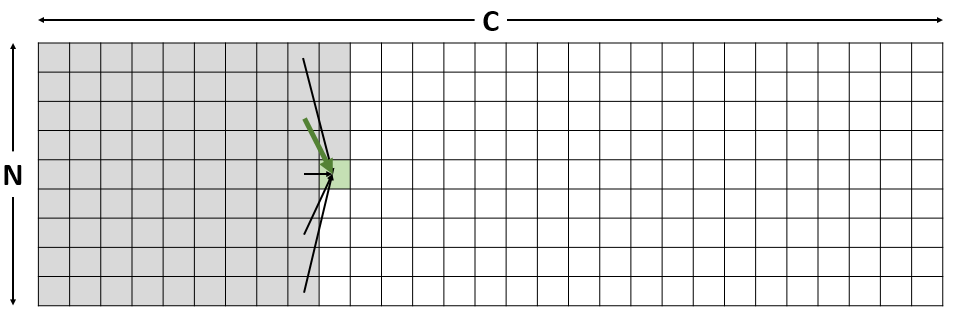


Figure 1. Customer/driver costs table for dynamic programming method

The advantage of dynamic programming method is that it allows to quickly get several suboptimal solutions, in case one of them cannot be used due to some restrictions.

Searching for alternative formulations

As a part of the research of the given problem we thought about various approaches with different techniques. We also discovered, that we can implement the dynamic programming algorithm without any assumptions to obtain the optimal value of the objective function. The algorithm would work as follows: for a given client, we would find the most appropriate insertion place among all the drivers and among all places in their assigned routes. This would give us the optimal solution in the end, but we can see that this algorithm will take inappropriately big amount of computation time since it is similar to brute-force algorithm. Also, the costs are not easily recomputed for a route when there are changes made in a middle of it.

Also, we tried to develop algorithms that consider 3 “dimensions” in our problem (i.e. clients, drivers and also times) but nevertheless, finding the optimal choice for inserting a client cannot be easily performed “locally” (without recomputations of costs of whole routes and affections on other drivers by the insertion), so the further research of dynamic programming application was abandoned.

* 1. Column generation

To try to tackle the problem in a different way we can try to reformulate our problem as a problem of finding the most efficient set of routes that delivers all our clients given that we know all such routes. Then we can reformulate our problem in a very simply way as a linear program:

where – cost of the routes, - a matrix contains the routes as columns, – decision variables where we decide whether we take a route or not (in this case, it is not binary, but the solution is *very likely* to become integer) and – the number of drivers.

However, considering the number of all different routes we can state that there are more than a lot such routes. To be exact, there are such routes so it becomes too costly to try to solve that problem with so many variables at a time.

The main insight of the column generation technique is that we do not need a lot of routes after all, as we can use a small "smart" subset of them and obtain an optimal solution (or a good solution) [1]. In the ideal case, we need to have only such routes (where is a number of drivers) to obtain the best possible solution for our problem. But actually, we do not have any routes initially and, moreover, we don't know what routes are the most effective.

Thus, column generation technique also implies solving a sub problem that helps us to find the most effective routes that we can use.

So, our column generation algorithm conceptually (can be done in more efficient way in terms of amount of computations performed) looks like the following:

1. Initialize the first set of the routes for the problem;

2. Solve the primal problem;

3. Identify dual variables for the dual problem;

4. Using dual variables, find a new route to add;

5. Exit the algorithm if a given time is exceeded or a new route is not an effective solution to the posed problem of the finding new route gives negative cost;

6. Go to step 2 after adding a new route to the existing set of routes.

For finding new routes to add to the primal problem we can use ESPPRC (Elementary Shortest Path Problem for Resource Constraints) [2] algorithm that will find the shortest paths on the routes given that we alter our weights of the graph using the dual variables from:

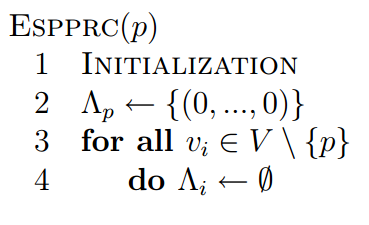
ESPPRC

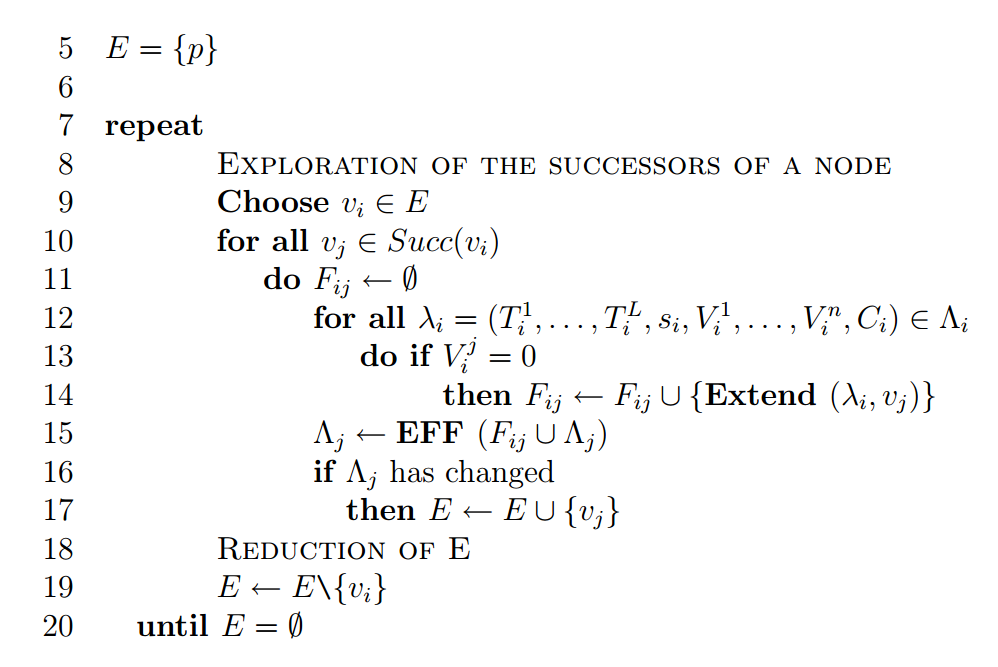
The basic overview of ESPPRC algorithm is the following. For each path at each node of the graph we have a structure called "label" that represents the entity of the path that is visited current node and also provides that information of the path that is going to this node, its cost and resources that are spent up to this point.

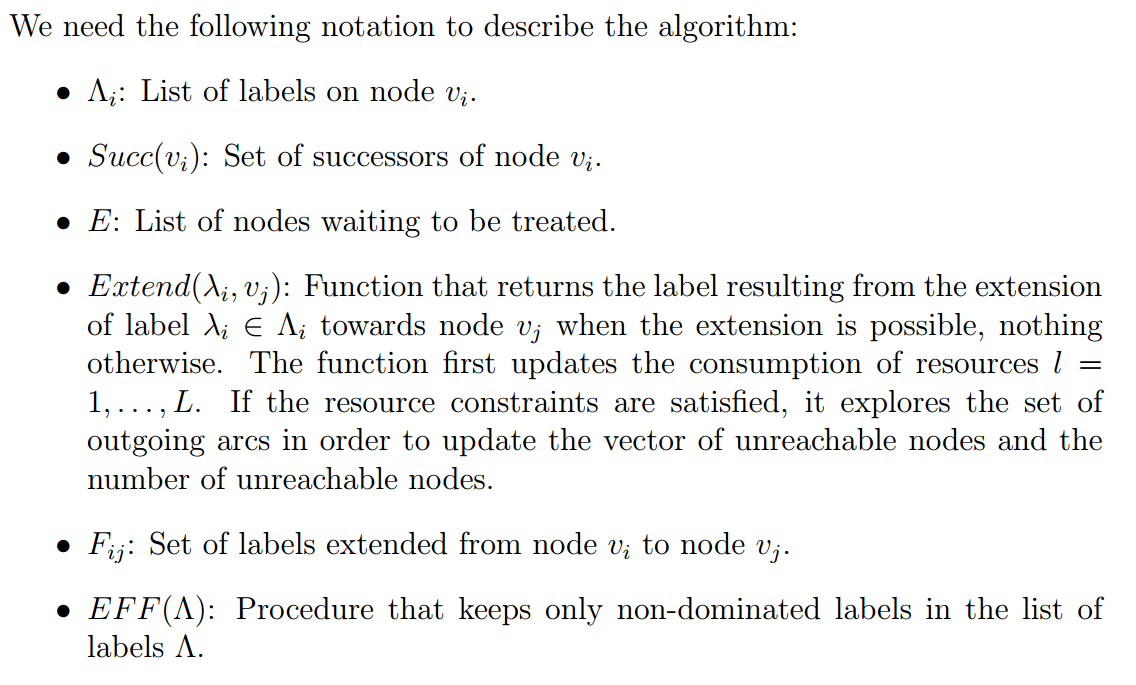
Also, we try to "extend" these labels further to the adjacent vertices, so we are actually extending the paths. Also, to reduce the growth of the amount of labels we provide a reduction based on the "dominance" of one label to another (this basically means, that we clearly see that a one label is more efficient than another so we can remove one from the problem).

In our problem, we try to obtain the best route from our depot point to the same depot (i.e. to itself). To make this possible, we provide 2 additional points on a graph that are placed exactly on our depot (in Euclidean distance sense) but we restrict any direct movement between these pairs of points. So, after that, our algorithm finds the path from the depot and back and also uses a subset of clients.

So we can provide a pseudocode of the algorithm, that is provided in the paper that we used in order to implement the algorithm:





Also we need to provide additional definitions of the pseudocode, to clarify some lines in the provided pseudocode: 

Results

Unfortunately, implementing the column generation technique is quite complex task considering the given time, so we didn't succeed to solve the initial problem using CG. We managed to implement ESPPRC algorithm and apply it to a simple small demo of our initial problem that obtains the best possible solution for the problem after some iterations.

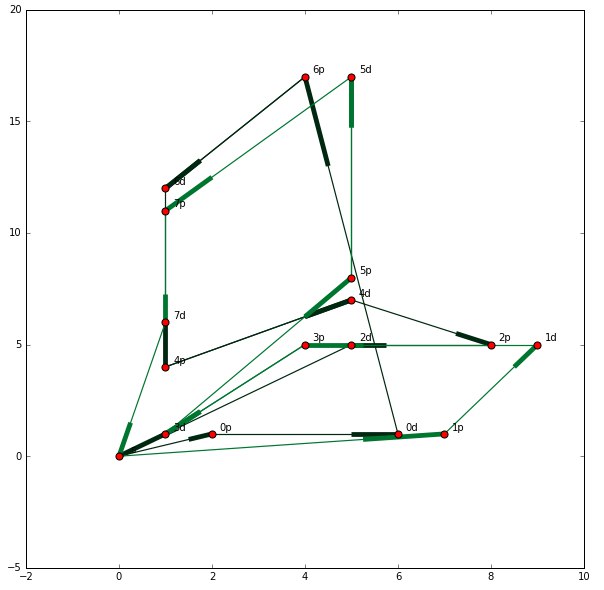


Figure 2. Graph of the first solution with initial set of specified routes for solver

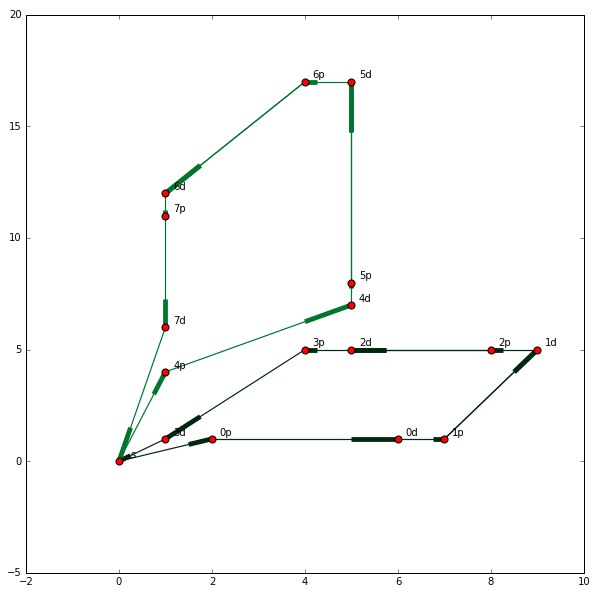


Figure 3. Graph of optimal taxi routes obtained by the column generation technique after generating additional routes to choose by solver

Nevertheless, solving the initial problem with the initial dataset becomes too computationally intensive, so we didn't provide any results on the bigger datasets.

More to say, we also tried to use Google Optimization Tools library [4] that can solve very similar problem of vehicle routing. We made an experiment of computing the vehicle routing problem on the of our dataset and we didn't manage to obtain result in a reasonable amount of time. Considering that solving this problem requires (way) non-linear time to compute we can state that solving the initial problem even with such powerful tool requires a lot of computations.

1. Data

Our team worked with Uber [dataset from Kaggle](https://www.kaggle.com/fivethirtyeight/uber-pickups-in-new-york-city) [5], containing information about Uber pickups in New York City on selected days in years 2014 and 2015. The following preprocessing steps were made:

* Only records for 4 Dec, 2014 were used;
* Half of the points were used as destination points for the other half;
* Latitude and longitude were converted into distance between points.

1. Results

Local search based on a random initial solution allows to get a decrease to random initial solution, but it is still worse than the greedy one, got with the greedy solution:

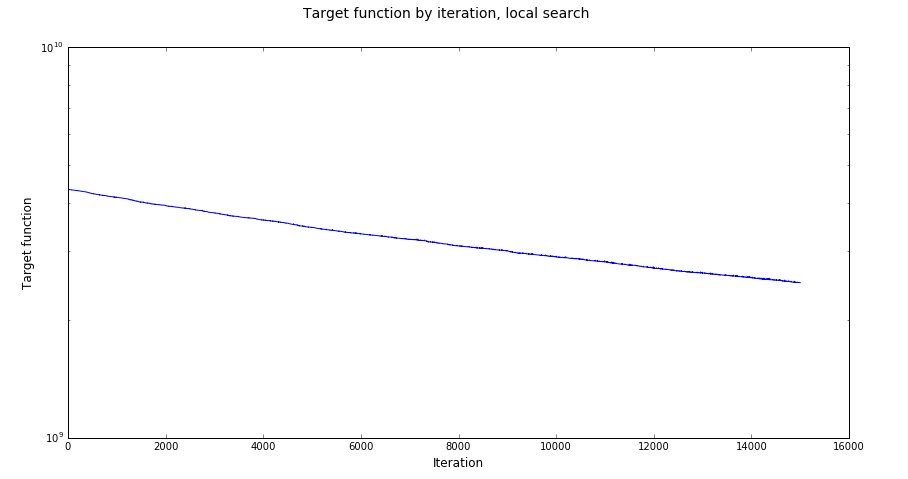


Figure 4. Target function by iteration for local search

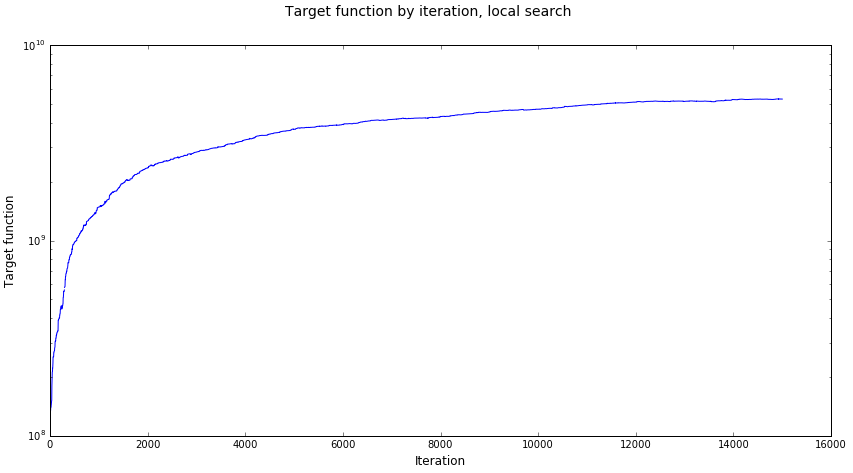


Figure 5. Target function by iteration for simulated annealing

Both local search and simulated annealing proved to be inferior to the greedy algorithm due to the stochastic nature of the problem. This is especially true for simulated annealing, as the probability to find a local step which reduces the target function is extremely low.

The quantitative results (target function values) for different methods applied to the dataset are as follows:

* Basic greedy algorithm: **1.3e+08**;
* Local search: **2.5e+09**;
* Simulated annealing: **5.3e+09**;
* (Greedy) Dynamic programming: **1.3e+08**.
* Column Generation: Unknown

Consider also, that numerical values should not be considered real-world costs of a taxi company (evaluated in US dollars or whatever) but is a conceptual measurement of costs. Providing more close to some real currency cost would take some additional time of research on taxi fares, gas prices and other topics which is not an objective of our work but we consider that this could be done as a possible extension of current work in future.

1. Conclusion

The greedy algorithm has proven to be the most feasible, as it has both the lowest computational time, and the lowest value of the target function.

Some further improvements to the implementation might be made:

* Adding a graph of New York City for higher accuracy of calculations;
* Adding variable fares, different taxi types and salaries for drivers;
* Adding greedy algorithm implementation, which works on data, partitioned by time and area to allow parallel computation.

Our implementation with result files, datasets and all code is available at public GitHub repository: <https://github.com/iwno/tcca>.

1. References
2. [Dominique Feillet – A tutorial on column generation and branch-and-price for vehicle routing problems, 4OR, 2010, Volume 8, Issue 4, pp 407–424](http://link.springer.com/article/10.1007/s10288-010-0130-z);
3. [Dominique Feillet, Pierre Dejax, Michel Gendreau, Cyrille Gueguen – An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems, Networks, Volume 44, Issue 3, October 2004, Pages 216–229](http://onlinelibrary.wiley.com/doi/10.1002/net.20033/abstract);
4. [Federico Liberatore, Giovanni Righini, Matteo Salani - A column generation algorithm for the vehicle routing problem with soft time windows, 4OR, March 2011, Volume 9, Issue 1, pp 49–82](http://link.springer.com/article/10.1007/s10288-010-0136-6);
5. [Vehicle Routing Problem with Time Windows - Google Optimization Tools – Developers. A guide for solving vehicle routing problem with time windows using OR tools](https://developers.google.com/optimization/routing/tsp/vehicle_routing_time_windows);
6. [Uber Pickups in New York City | Kaggle](https://www.kaggle.com/fivethirtyeight/uber-pickups-in-new-york-city).