

USER EQUILIBRIUM SOLUTION IN TRAFFIC ASSIGNMENT

USING FRANK-WOLFE ALGORITHM

Team:

- 1) Amogha | 150040102
- 2) Priyharsh | 150040089
- 3) Ashwini K | 150040040
- 4) Ayush | 150040087

Introduction: Traffic assignment or trip assignment is the last step of classical four step model of travel demand modelling. In this step we allocate a given set of trip interchanges to specified transportation system. The fundamental aim of this process is to reproduce on the transportation system, the pattern of vehicular movements which would be observed when the travel demand represented by the trip matrix, or matrices, to be assigned is satisfied.

There are different types of traffic assignment models such as all-or-nothing assignment (**AON**), incremental assignment, capacity restraint assignment, deterministic user equilibrium assignment (**UE**), stochastic user equilibrium assignment (**SUE**), system optimum assignment (**SO**), etc

In our project we have focused on **Deterministic User Equilibrium (UE)** method which is a widely used model among various available trip assignment models.

Deterministic User Equilibrium Model - Deterministic User Equilibrium model of traffic assignment is based on the fact that humans choose a route so as to minimize his / her travel time and on the assumption that such a behaviour on the individual level creates an equilibrium at the system (or network) level.

In terms of Wardrop's first principle - Flows on the links (whose travel time vary with the flow) are said to be in equilibrium when no driver/trip maker can unilaterally reduce his/her travel costs by shifting to another route.

Since this is a deterministic model it assumes the fact that drivers are completely rational and have knowledge of all travel cost and travel time on a given link is a function of flow on that link only.

Mathematical Formulation -

$$\text{Minimize } Z = \sum_a \int_0^{x_a} t_a x_a dx$$

Subjected to:

$$\sum_k f_k^{rs} = q_{rs} : \forall r, s$$

$$x_a = \sum_r \sum_s \sum_k \delta_{a,k}^{rs} f_k^{rs} : \forall a$$

$$f_k^{rs} \geq 0 \quad \forall k, r, s$$

$$x_a \geq 0 : a \in A$$

where: x_a – equilibrium flows in link a,

t_a – travel time on link a , $t = t_0 \left\{ 1 + \alpha \left(\frac{x}{k} \right)^\beta \right\}$, where α and β are specific to type of link and needs to be calibrated from field data.

f_k^{rs} – flow on path k connecting O-D pair

q_{rs} – Trip rate between r and s

$\delta_{a,k}^{rs} = 1$ if link 'a' belongs to path 'k'
 0 otherwise

The equation is simply the flow conservation equation and non negativity constraint respectively. These constraints naturally hold the point that minimizes the objective function. The path connecting O-D pair can be divided into two categories : those carrying the flow and those not carrying the flow on which the travel time is greater than (or equal to) the minimum O-D travel time. If the flow pattern satisfies these equations no motorist can better off by unilaterally changing routes. All other routes have either equal or heavy travel times. The user equilibrium criteria is thus met for every O-D pair.

This problem is convex optimization problem because the link travel time functions are monotonically increasing function, and the link travel time a particular link is independent of the flow and other links of the networks.

To solve such problem we have used Frank Wolfe Algorithm -

Frank-Wolfe Algorithm is used for solving quadratic programming problem with linear constraints. It is applicable to nonlinear programming problems with convex objective functions.

The algorithm involves two key implementation steps which are -

- 1) **Descent direction search** : Maximize the *drop* (product of the rate of descent in a given direction and the length of feasible move in that direction); popular methods include linear programming and Dijkstra's shortest path algorithm.

- 2) **Line search** : Since the bounding of the move size is naturally accomplished in the descent direction search process, simple interval reduction methods could be used to find the optimal step size.

The above steps are implemented repeatedly until a convergence is reached

We have used Python to replicate the process described above in the following steps (for more details, please refer to the comments in the code):

- 1) Use array data structures to represent the network and given information;

Input Datasets	Python Variable	Definition	Data Structure	Size
Network Information: Link & Node	n, k	n: # of links k: # of nodes	Python number	1
Network Information: Link & Node	Link Node	Link-node Incidence matrix (adjacency)	Python array	nparray(n, k)
OD flow demand (q_{ij})	Q	Travel flow demand between each OD pair	Python array	n parray(n,n)
Link travel time information: free Flow time,parameters for the BPR function,link capacity	coeff	Includes two columns, column 1 is for free flow travel time on each link (t_0); column 2 is for link capacity (ca), other parameters used in the BPR function (e.g., α ,	Python array	nparray(n,2)

		beta) are constants		
--	--	---------------------	--	--

- 2) Conduct the descent direction search by forming a Linear Programming problem and solve it;
 - **Initialization:** carry out the All or Nothing flow assignment (find x_a) based on the free flow travel time on each link (t_0), then update the travel time on each.
 - **Iteration (direction search):** Formulate the objective function (Z) and constraints for the LP program, then use the `*optimize.linprog*` function in Python package to solve the problem and obtain updated link flow (y_a)
- 3) Find the optimal step size using the Golden section interval reduction method for the one-dimensional search;
 - **Iteration (move):** formulate the line search function and use the `*minimize_scalar*` function in Python package to solve the step size α
- 4) Update the network and check for convergence.
 - **Iteration (update):** update link flow (x_a) using y_a and α , then use x_a to update t_a .
 - **Iteration (check convergence):** Calculate the norm of (iteration n) and t_a (iteration $n-1$), when $\text{Norm} < 0.1 \cdot n$ (on average the link flow changes 0.1), the iteration stops and the algorithm is converged

```

90
91 b0 = np.transpose(RHT)
92 b = np.ravel(b0, order = 'F')[:,np.newaxis]
93
94 ybounds = (0, None)
95 result = optimize.linprog(
96     c_0, A_eq = A, b_eq = b, bounds
97 )
98 #print result
99 #print len(result['x'])
100 result = np.reshape(result['x'], (k, r))
101 #print result
102 xa = np.sum(result, axis = 0) # int
103 #print xa
104 ta = firstfunc(t0, xa, ca)
105
106
107 #####
108 step = 0
109 tanorm = 1000000
110
111 iteration = []
112 Z = []
113
114 while (tanorm > (n/10)): # allow each
115     ### Update
116     print ("step ", step)
117     iteration.append(step)
118     ta_old = ta
119

```

Command Prompt - python help.py

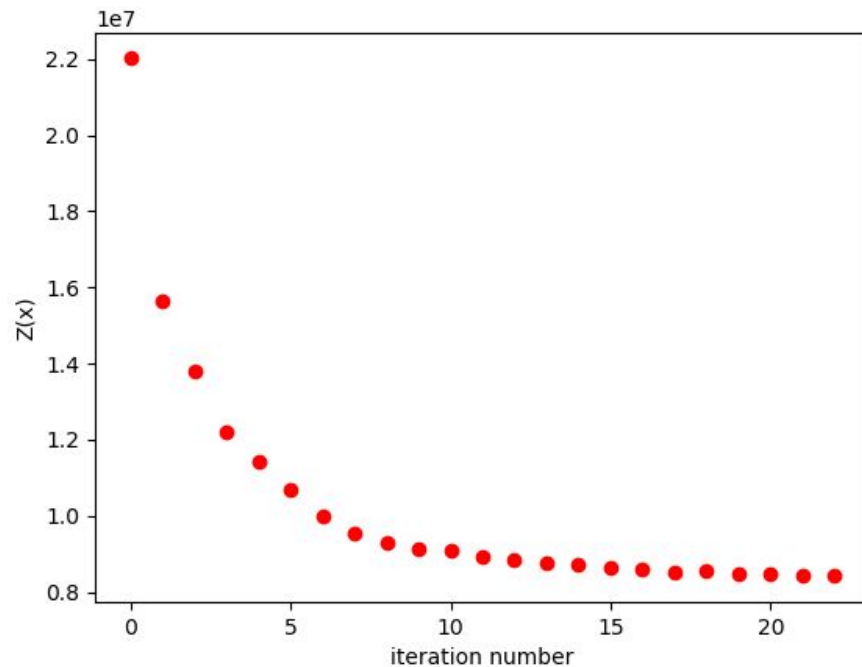
```

10792.47660319 8415.27908555 16291.9633872 10362.74969253
23694.48005844 22819.52845243 18001.59100845 24690.63772317
12482.50987066 10557.61544012 7450.31541802 17930.32021264
8643.22345745 12685.36567131 10532.13401303 9244.58064249
13954.77537789 14083.8063396 11557.19433197 12067.39761497
8969.93871498 10005.71646209 24057.26114658 8816.15976631
21513.63430389 20199.541542 9140.31636644 11971.28079746
15693.97754599 17908.22439065 10798.54027971 15457.06317569
11708.94764633 17143.87057492 18430.95296864 17656.64388486
21739.69901001 11712.95811561 10974.71976987 18974.94708679
11204.79494528 9540.99866062 12442.31817324 9766.1576357
11745.44375023 10125.80504718 19086.32131061 13665.53757551
12513.59321927 10353.6155087 9541.52735443 11231.76914562
9965.83936111 11586.22539368 9580.8005321 10379.80321337]
ta is [ 6.9000814  4.00739369 6.02250611 5.6732792 4. 4.57050508
4.03641024 4.06183953 2.73530235 10.89294315 2.15989735 52.27527577
11.02753189 36.27907936 8.76963873 14.99824143 9.71105167 2.06847252
21.06435742 8.1777702 41.28554799 10.80238863 10.28390457 36.59251026
6.78242198 6.25392778 12.875984 16.03443659 30.21992397 31.97842559
10.77562631 12.75199432 14.65036958 31.47434499 4.02460909 17.3208897
3.03792201 3.03934415 19.93166341 26.499345 12.02405076 14.22334317
15.04374887 11.55462041 5.1421759 11.82352093 13.07566454 26.18126071
26.32632364 3.3135463 34.24324405 24.89034301 12.41312835 2.08638444
3.34618833 4.19438495 5.23364478 12.42740221 17.89758273 4.25927001
19.10014652 17.37747466 32.08218217 18.49008682 9.63171406 11.29854942
10.03339151 44.40808454 11.8327093 15.0316218 12.45418546 19.27790859
6.4486893 20.09234502 9.65634649 7.23519833]
norm of ta is 89.07810873960733
step 3

```

Results of the Implementation-

- Two csv files are generated as results.
- The graph of number of iterations and the objective function value is being attached here.



The above algorithm is deterministic in nature assumes that drivers are completely rational and identical and have complete knowledge of flows and networks This is usually not the case in reality and this is the reason why these models fall short in giving a proper estimate for any arbitrary network.