CEGE 3201: Static traffic assignment in Python

Michael W. Levin

October 27, 2024

1 Introduction

The purpose of this lab is to implement a basic traffic assignment algorithm to better understand how computers are used to solve traffic assignment. This lab is written in Python 3, so you will first need to set up some version of Python 3 on your computer to run it. (Note that Python versions 1 and 2 are not compatible with this lab due to the object-oriented code.) This lab will list specific exercises of code implementation that are designed to be completed sequentially as they build on the code written previously.

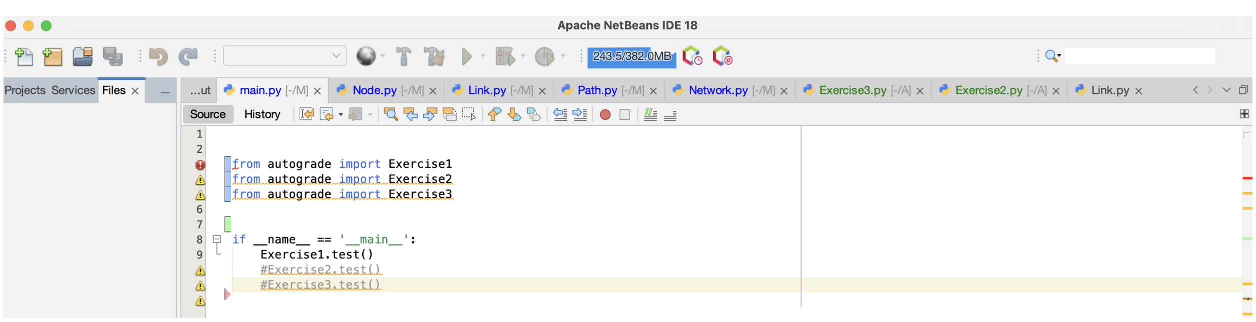
Download a copy of this Git repository: https://github.com/mwlevin/STA_python_3201.git. The repository contains 4 folders.

- The "documentation" folder contains this explanation.
- The "src" folder contains the source files you will use and modify.
- The "autograde" folder contains code to automatically check the correctness of your work.
- The "data" folder contains some test data that will be used within this lab. If you are curious, you can find data for different cities in the same format at

  https://github.com/bstabler/TransportationNetworks

  After you complete this lab, you will be able to solve traffic assignment on other networks (although your algorithm will be very basic, so it may take a long time).

The `main()` function (which is executed when you run the program) is found in main.py. Each of the exercises in this tutorial are contained within a separate file, e.g. Exercise1.py, Exercise2.py, etc. Each of these files has their own `test()` function that can be executed. In main.py, you will find calls to these functions commented out, i.e. `#Exercise1.test()`. Uncomment them to run each exercise.



Each exercise contains some test code. At the end of the `test()` function, it calls the `autograde()` function which will automatically test the output of your code against the correct answers. Go ahead and run the code; it should run successfully, but the autograde will indicate that your code is incorrect:

```
STA_python_3201 — -zsh — 147×24
michael@MacBookAir STA_python_3201 % python3.8 main.py
link 1 TT with flow 1320.2 0.0
link 2 TT with flow 570 0.0
link 1 TT with flow 0 0.0
link 2 TT with flow 2512 0.0
TSTT 0


********************
*    Autograde     *
********************
Testing 1(a): Link.getTravelTime(): 0 / 4 (0%)
Testing 1(b): getTSTT(): 0 / 1 (0%)

Total: 0 / 5 (0%)

********************
*  End Autograde   *
********************
michael@MacBookAir STA_python_3201 %
```

2 Calculating travel times

We will use (i, j) to refer to a link from node i to node j, with $\mathcal{A}$ the set of all links.

Exercise 1(a) Implement the calculation of the link travel time $t_{ij}(x_{ij})$ using the BPR function

$$t_{ij}(x_{ij}) = t_{ij}^{\text{ff}} \left(1 + \alpha_{ij} \left(\frac{x_{ij}}{C_{ij}}\right)^{\beta_{ij}}\right) \tag{1}$$

where t_{ij}^{ff} is the free flow travel time, C_{ij} is the link capacity, and α_{ij} and β_{ij} are calibration constants. You will see some variables `x`, `t_ff`, `C`, `alpha`, and `beta` defined in the function `__init__()`. These variables are available for use anywhere within the `Link` class by using the "`self`" reference, and they contain the values you need. (These values have either been specified or read from one of the data files.)

Within the `Link` class in Link.py, find the function labeled `getTravelTime()`. It defines a variable `t_ij` and sets the value to 0.0 — a floating-point number. You need to calculate the correct value of $t_{ij}(x_{ij})$ and assign it to variable `t_ij`.

It may help to look at Python syntax, variables and data types. For math calculations, it may help to look at tutorials on operators and the math functions.

```python
# **********
# Exercise 1
# **********
def getTravelTime(self):
    t_ij = 0.0
    # fill this in
    return t_ij
```

Open main.py and ensure that it will run `Exercise1.test()`. Open autograde/Exercise1.py. The `main()` method constructs two instances of the `Link` class with different parameters. The first link has $t_1^{\text{ff}} = 10$, $C_1 = 2580$, $\alpha_1 = 0.15$, and $\beta_1 = 4$. The second link $t_2^{\text{ff}} = 12$, $C_2 = 1900$, $\alpha_2 = 0.35$, and $\beta_2 = 2$. The `main()` function then prints the calculation of t_{ij} with $x_1 = 1230.2$, $x_2 = 570$, $x_1 = 0$, and $x_2 = 2512$. You should compare the values calculated by your code with values that you have computed by hand. Afterwards, `test()` calls the `autograde()` function, which runs an automated test of your answers.

Exercise 1(b) Open Network.py and implement the `getTSTT()` function to return the total system travel time. The TSTT is defined as

$$TSTT = \sum_{(i,j)\in\mathcal{A}} x_{ij} t_{ij}(x_{ij}) \tag{2}$$

The value for x_{ij} is available from a variable in the Link class, and you wrote the method for $t_{ij}(x_{ij})$ in Exercise 1(a). It may be helpful to look at tutorials on `if` statements and for loops here. Also, we are working with object-oriented programming, so it may be helpful to read the tutorial on classes to understand the interactions between the code. We will not ask you to implement any specific object-oriented code, but you will be working with `Node`, `Link`, and `Path` objects.

After completing Exercises 1(a)–1(b), your code should pass the `autograde()` method of Exercise1.py.

3 Dijkstra's shortest path algorithm

We need to implement a shortest path algorithm, which will be used as a step in solving traffic assignment. We will implement the well-known Dijkstra's algorithm, which finds the one-to-all shortest path. We need to define two variables for this. Let $V(n) \in \mathbb{R}_+$ be the cost label of node n, and let $P(n) \in \mathcal{N}$ be the predecessor node. First, read through a pseudocode of this algorithm, which we also saw in class:

procedure DIJKSTRA'S(r)
    for $n \in \mathcal{N}$ **do** ▷ Initialization
        $V(n) \leftarrow \infty$
        $P(n) \leftarrow \emptyset$
    end for
    $c_r \leftarrow 0$
    $Q \leftarrow \{r\}$
    while $Q \neq \emptyset$ **do** ▷ Main loop
        $u \leftarrow \arg\min_{n\in Q} \{V(n)\}$
        $Q \leftarrow Q/\{u\}$
        for $(u, v) \in \mathcal{A}$ **do**
            if $c_u + t_{uv} < c_v$ **then** ▷ Is this a shorter path to v?
                $c_v \leftarrow c_u + t_{uv}$ ▷ If so, update v and add it to Q

14: $p_v \leftarrow u$
15: $Q \leftarrow Q \cup \{v\}$
16: **end if**
17: **end for**
18: **end while**
19: **end procedure**

This may be your first time implementing pseudocode, so we will break it down into steps. The first is the initialization. In line 2, we start looping through all nodes in set $\mathcal{N}$. Within this loop, set $V(n) \leftarrow \infty$. The operator $\leftarrow$ is used to indicate that $V(n)$ is assigned the value ∞, which exists in python. $P(n)$ is assigned the value $\emptyset$, or `None` in Python, i.e. $P(n)$ is initialized to not be any specific node. After the loop, in line 6 we set $c_r \leftarrow 0$. Recall that r is the origin parameter to Dijkstra's, so r is the starting point. Therefore the shortest path from r to r has cost 0. Finally, in line 7 we construct the set $Q \subseteq \mathcal{N}$ which contains the unsettled nodes.

Next, we enter the main loop in line 8. This loop continues while Q is non-empty — while there is an unsettled node that we need to visit. Line 9 is written very simply, but can actually require more extensive code. Finding the $\arg\min_{n \in Q} \{V(n)\}$ could involve looping through all elements of Q to find the n with the smallest value of $V(n)$. Save that node and store it in variable u. Once you have determined u, remove it from Q. Then loop through all outgoing links (u, v) in line 11. The function `getOutgoing()` of the `Node` class which you implemented previously will be useful here. In line 12, notice that while c_u and c_v will be variables, t_{uv} is a function call to `getTravelTime()` of the `Link` class. Line 15 requires adding node u to set Q. Beware of adding multiple copies of u to your implementation of Q, which is possible with some data structures (such as lists). Instead, use a python set to implement the set Q. If done correctly, Q will eventually become empty, and the algorithm will terminate after calculating $V(n)$ and $P(n)$ for all nodes.

We will start our implementation of Dijkstra's by implementing a data structure to store a path. A `Path` is an ordered list of `Links`. Open Path.py. A `Path` contains a list of links and includes the following functions:

- `add()`: add a link to the end of the list.
- `addFront()`: add a link to the front of the list.

You will need some of these when constructing a `Path` to store the shortest path. The `Path` class is useful because it defines five additional functions to perform calculations on the path:

- `isConnected()` checks whether the list of links is a valid path. For instance, the list `[(1,3), (3, 7), (7, 8)]` is a connected path, but the list `[(1,3), (2, 4), (4, 8)]` is not.
- `getSource()` and `getDest()` return the origin and destination nodes of the path, respectively.
- `getTravelTime()` calculates the travel time for the path.
- `addHstar()` will be used later.

Exercise 2(a)

- Open Path.py. Implement the `getTravelTime()` function, which returns $T^\pi = \sum_{(i,j)\in\pi} t_{ij}(x_{ij})$. Intuitively, the travel time on the path is the sum of the travel times of the links that comprise the path. You will need to use the link travel time function from Exercise 1(a).

To implement Dijkstra's, we need two additional variables $V(n)$ and $P(n)$. Open Node.py. You will see that the instance variables `cost` (representing $V(n)$) and `predecessor` (representing $P(n)$) have already been created for you.

Exercise 2(b) Open Network.py and navigate to the `dijkstras()` function. Implement the initialization (lines 2–7) of Dijkstra's algorithm. You may wish to test the correctness of the initialization using the autograde before proceeding further. It may be helpful to look at the `None` type in Python, which indicates an object reference that currently does not refer to anything.

Exercise 2(c) In Network.py, implement the main loop of Dijkstra's algorithm (lines 8–18) in the `dijkstras()` function.

After executing Dijkstra's algorithm, we now have all the information needed to find the shortest path from r to s through the predecessor labels. We need to convert those predecessor labels into an instance of the `Path` class created earlier. This can be accomplished through the trace algorithm shown below. Essentially, start at s, and follow the predecessor labels until reaching r, adding each link to the path as you go.

procedure TRACE(r, s)
    $n \leftarrow s$
    $\pi \leftarrow \emptyset$
    while $n \neq r$ **do**
        $\pi \leftarrow \pi \cup \{(P(n), n)\}$
        $n \leftarrow P(n)$
    end while
end procedure

Exercise 2(d) Open Network.py. Implement the `trace()` function in the `Network` class. It may be useful to add the links in the correct order to ensure a connected path, which can be checked afterwards by the `isConnected()` function of the `Path` class.

After completing Exercises 2(a)–2(d), your code should pass the `autograde()` method of Exercise2.py.

4 Method of successive averages for traffic assignment

The method of successive averages is a simple algorithm for solving user equilibrium. Each iteration, it constructs an all-or-nothing flow assignment $\mathbf{x}^\star$ formed by assigning all flow from r to s to the shortest path from r to s. Then, it takes a weighted average between the current and the all-or-nothing flow assignment. The weight, or step size, is denoted by λ. This step is repeated until the maximum number of iterations, I, is reached. We can track the convergence towards user equilibrium by printing the average gap between vehicle travel times and the shortest path travel time each iteration. The algorithm is specified below in pseudocode:

1: **procedure** METHOD OF SUCCESSIVE AVERAGES(I)
2:     **for** $(i, j) \in \mathcal{A}$ **do** ▷ Initialization
3:         $x^{\star}_{ij} \leftarrow 0$
4:     **end for**
5:     **for** *iteration* $\leftarrow 1$ to I **do**
6:         **for** $r \in \mathcal{Z}$ **do**
7:             DIJKSTRA'S(r) ▷ Find shortest paths from r to s
8:             **for** $s \in \mathcal{Z}$ **do**
9:                 $\pi^{\star}_{rs} \leftarrow$ TRACE(r, s)
10:                 **for** $(i, j) \in \pi^{\star}_{rs}$ **do** ▷ Update all-or-nothing flow assignment
11:                     $x^{\star}_{ij} \leftarrow x^{\star}_{ij} + d_{rs}$
12:                 **end for**
13:             **end for**
14:         **end for**
15:         $\lambda \leftarrow \frac{1}{iteration}$ ▷ Calculate step size
16:         **for** $(i, j) \in \mathcal{A}$ **do** ▷ Take weighted average between $\mathbf{x}$ and $\mathbf{x}^{\star}$
17:             $x_{ij} \leftarrow (1 - \lambda)x_{ij} + \lambda x^{\star}_{ij}$
18:             $x^{\star}_{ij} \leftarrow 0$
19:         **end for**
20:         PRINT(AEC) ▷ Track convergence
21:     **end for**
22: **end procedure**

Exercise 3(a)

- Open Network.py. Implement the `calculateAON()` function in the `Network` class, which is the loop in lines 6–14.

Exercise 3(b)

- Open Link.py. Notice that there is an instance variable to store $x^{\star}_{ij}$ in the `Link` class and a `addXstar()` function which adds the specified flow to the $x^{\star}_{ij}$ variable. It will be used to implement line 11.
- Open Link.py. In the `Link` class, implement the `calculateNewX()` function, which takes as input λ and implements lines 17 and 18.
- Open Network.py. In the `Network` class, implement the `calculateStepsize()` function, which determines the value of λ in line 15.

- Open Network.py again, and implement the `calculateNewX()` function of the `Network` class, which implements the loop in line 16 by calling the `Link.calculateNewX()` function on every link in the network.

Exercise 3(c) In the method `msa(int)` of the `Network` class, check that the method of successive averages succeeds. I already wrote part of the code for you, but you need to implement the calculations per iteration by using the methods you wrote in Exercises 3(a) and 3(b).

After completing Exercises 3(a)–3(c), your code should pass the `autograde()` method of Exercise3.py.