

# A tutorial for programming static traffic assignment in Python

Michael W. Levin

September 8, 2023

## 1 Introduction

The purpose of this tutorial is to guide you through learning the programming concepts necessary to implement the method of successive averages for solving user equilibrium. (For more information on user equilibrium, see [Transportation Network Analysis](#).) This tutorial consists of a series of programming exercises that increase in difficulty and required programming knowledge. After completing all exercises, you will have a working implementation of the method of successive averages. To assist with these exercises, I have linked relevant programming tutorials and provided an autograde to check correctness. Some code is provided as a starting point. This tutorial and the code is based on the Python programming language and object-oriented programming.

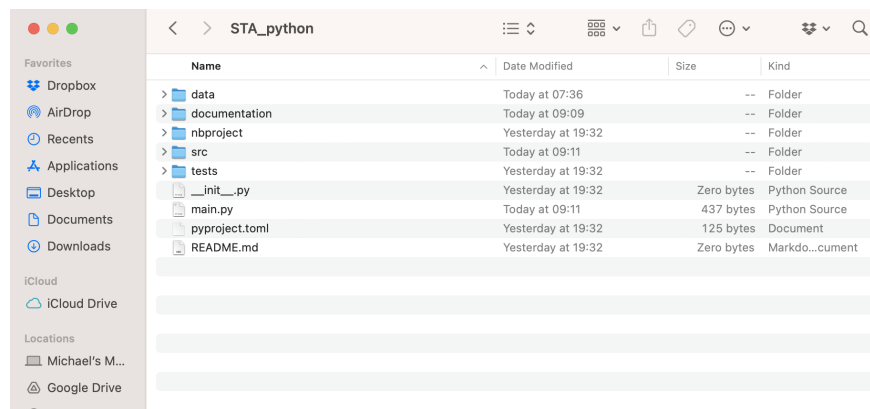
### 1.1 Getting started

The existing code is provided as a Netbeans project. You may use Netbeans or an alternative of your choice.

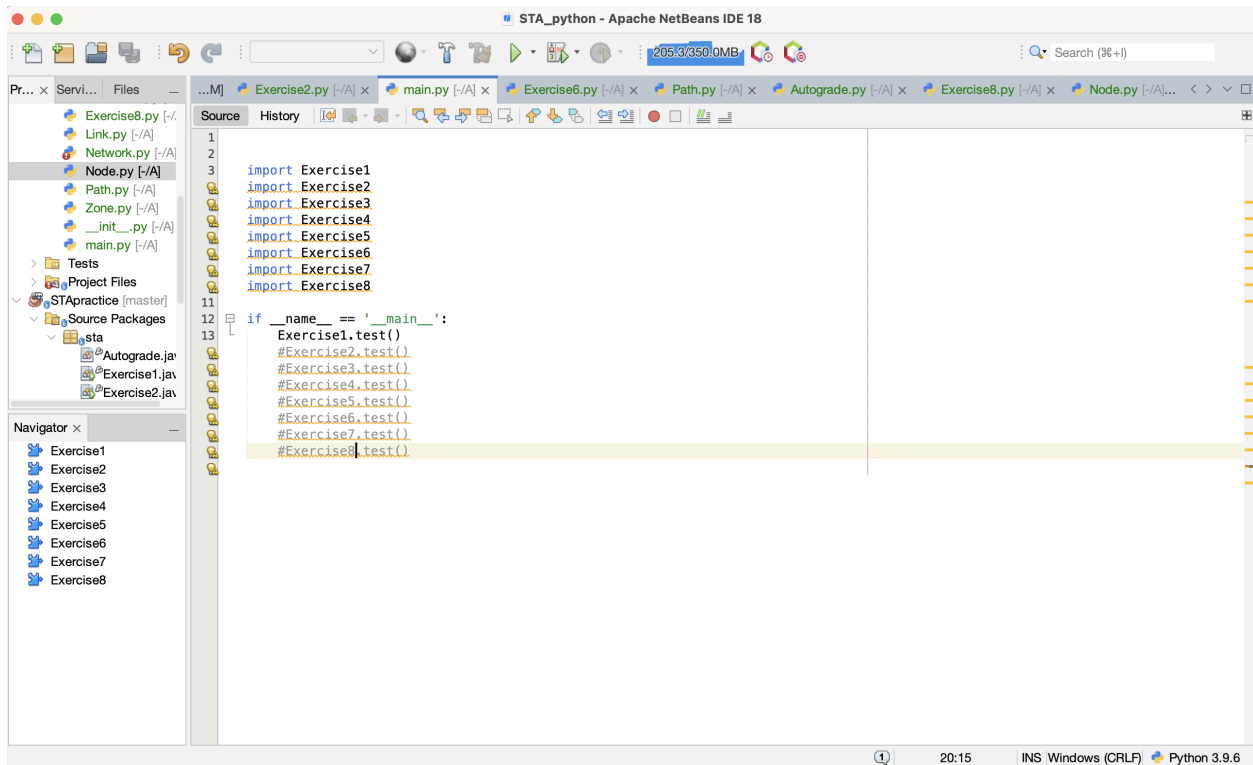
**To use Netbeans:** Download and install [Netbeans](#). Follow the [instructions here](#) to install the Python plugin. Download a copy of this Git repository:

<https://github.com/mwlevin/STApractice-python.git>. Alternatively, you can [clone it in Netbeans](#).

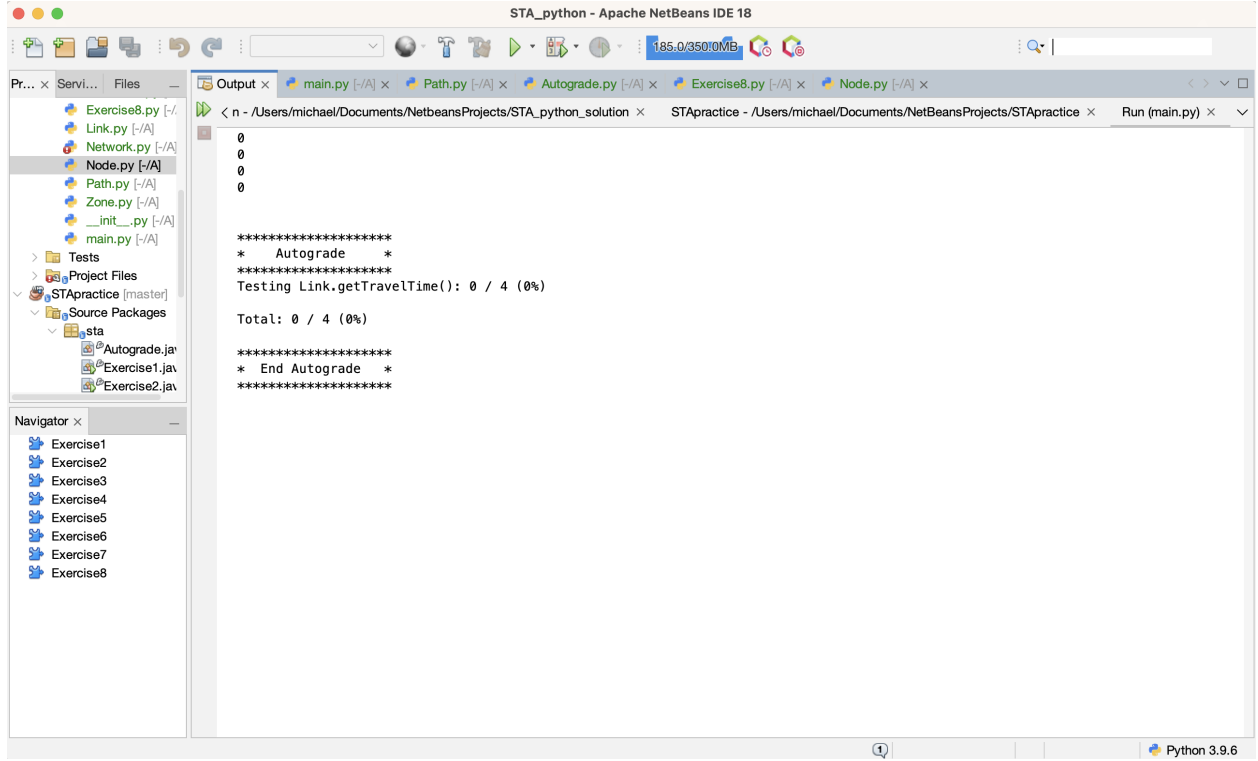
The repository contains a Netbeans project, which you may open directly in Netbeans. You can also open the source files in another IDE. All source files are in the “src” folder, with the exception of main.py. Test data is in the “data” folder:



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:



## 1.2 Notation

This section defines the notation for the traffic assignment problem being solved. For more details on the definition, see [Transportation Network Analysis](#). Consider a network  $\mathcal{G} = (\mathcal{N}, \mathcal{A})$  with nodes  $\mathcal{N}$  and links  $\mathcal{A} \subseteq \mathcal{N}^2$ . Let  $\Gamma_i^+ \subseteq \mathcal{A}$  be the set of links outgoing from node  $i$ . The travel time  $t_{ij}$  for link  $(i, j) \in \mathcal{A}$  is a function of the flow on that link  $x_{ij}$ , and is given by the BPR function

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

where  $t_{ij}^{\text{ff}}$  is the free flow travel time,  $C_{ij}$  is the link capacity, and  $\alpha_{ij}$  and  $\beta_{ij}$  are calibration constants.

Let  $\mathcal{Z} \subseteq \mathcal{N}$  be the set of zones. All trips start and end at zones. The demand from zone  $r$  to zone  $s$  is denoted as  $d_{rs}$ . A path  $\pi$  consists of a set of links. Let  $\Pi$  be the set of all paths, and let  $\Pi_{rs} \subseteq \Pi$  be the set of paths from  $r$  to  $s$ . Let  $h^\pi$  be the flow on path  $\pi$ , and let  $T^\pi$  be the travel time for path  $\pi$ . Let  $\delta_{ij}^\pi \in \{0, 1\}$  indicate whether path  $\pi$  includes link  $(i, j)$ . Then  $T^\pi$  can be written as

$$T^\pi = \sum_{(i,j) \in \mathcal{A}} \delta_{ij}^\pi t_{ij}(x_{ij}) \quad (2)$$

The user equilibrium problem is to find a path flow assignment  $\mathbf{h}$  such that

$$h^\pi (T^\pi - \mu_{rs}) = 0 \quad (3)$$

where  $\mu_{rs}$  is the minimum travel time from  $r$  to  $s$ . The solution can be found by solving the convex program

$$\min \quad Z = \sum_{(i,j) \in \mathcal{A}} \int_0^{x_{ij}} t_{ij}(\omega) d\omega \quad (4a)$$

$$\text{s.t.} \quad x_{ij} = \sum_{\pi \in \Pi} \delta_{ij}^{\pi} h^{\pi} \quad \forall (i,j) \in \mathcal{A} \quad (4b)$$

$$d_{rs} = \sum_{\pi \in \Pi_{rs}} h^{\pi} \quad \forall (r,s) \in \mathcal{Z}^2 \quad (4c)$$

$$h^{\pi} \geq 0 \quad \forall \pi \in \Pi \quad (4d)$$

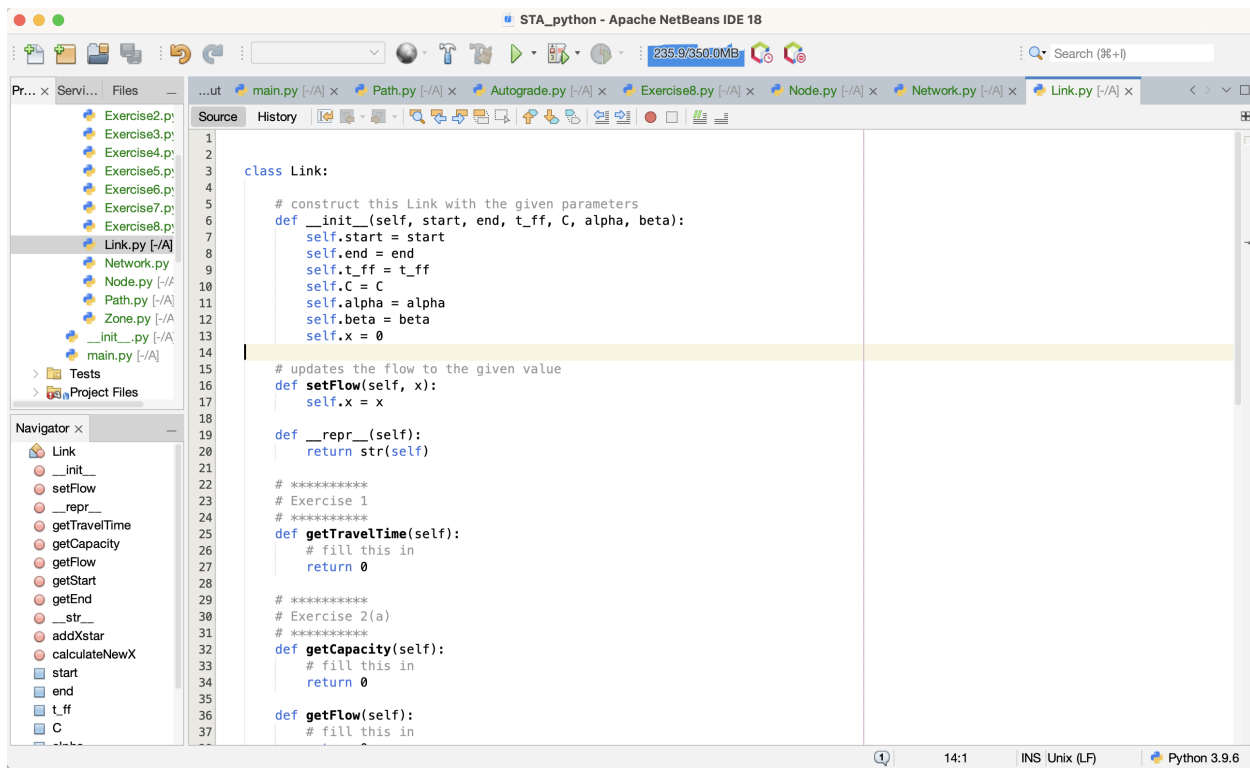
This tutorial will guide you through the steps needed to implement the method of successive averages algorithm for solving this problem.

## 2 Primitive data types, control logic, and arrays

### 2.1 Calculating link travel times

First, review [Python syntax](#) and [comments](#). Read the tutorials on [variables](#) and [data types](#).

Open `Link.py`. You will notice that the code first defines a `class Link`, which is a new data type intended represent one link  $(i,j) \in \mathcal{A}$ . Each  $(i,j)$  should have a separate instance of the `Link` class. We will learn later about creating and working with classes in Python. You will see some variables `x`, `t_ff`, `C`, `alpha`, and `beta` defined in the function `__init__()`. For now, it is sufficient to know that these variables are available for use anywhere within the `Link` class by using the “`self`” [reference](#). These variables correspond to the model variables  $x_{ij}$ ,  $t_{ij}^{\text{ff}}$ ,  $C_{ij}$ ,  $\alpha_{ij}$ , and  $\beta_{ij}$  for the specific link  $(i,j)$  being represented.



Read the tutorials on [operators](#) and the [math functions](#).

The tutorial will list specific exercises of code implementation that are designed to be completed sequentially as they build on the code written previously. The autograder may not be able to check correctness if you complete them out of order. We are now ready for the first exercise.

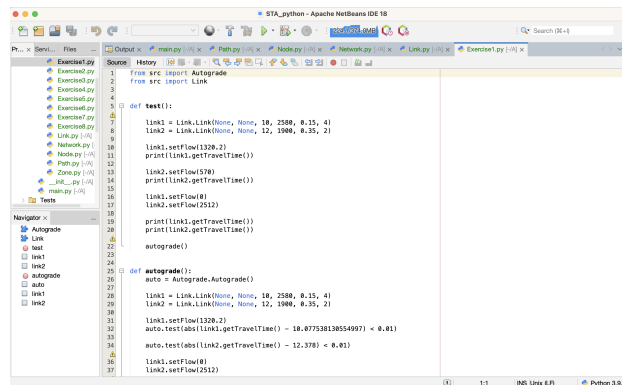
**Exercise 1** Your first task is to implement the calculation of the link travel time  $t_{ij}(x_{ij})$  using the BPR function of equation (1). Assume that the values of  $x_{ij}$ ,  $t_{ij}^{\text{ff}}$ ,  $C_{ij}$ ,  $\alpha_{ij}$ , and  $\beta_{ij}$  are already given.

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`.

```
# *****
# 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 `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.

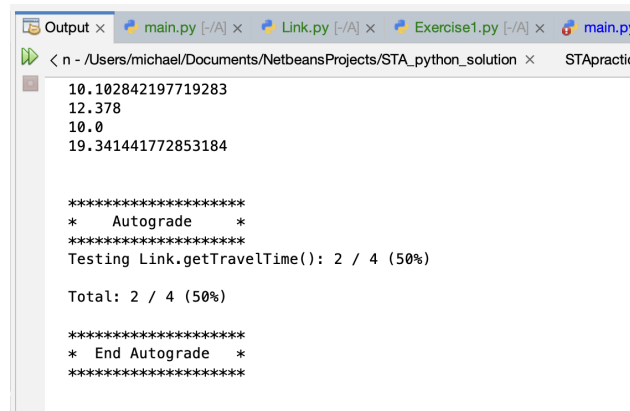


```

1  from src import Autograde
2
3
4
5  def test():
6      link1 = Link(LinkName, None, 18, 2500, 0.15, 4)
7      link2 = Link(LinkName, None, 12, 1000, 0.35, 2)
8
9
10     link1.setFlow(1328.2)
11     print(link1.getTravelTime())
12
13     link2.setFlow(578)
14     print(link2.getTravelTime())
15
16     link1.setFlow(8)
17     link2.setFlow(2512)
18
19     print(link1.getTravelTime())
20     print(link2.getTravelTime())
21
22     autograde()
23
24
25  def autograde():
26     auto = Autograde.Autograde()
27
28     link1 = Link(LinkName, None, 18, 2500, 0.15, 4)
29     link2 = Link(LinkName, None, 12, 1000, 0.35, 2)
30
31     link1.setFlow(1328.2)
32     auto.testAuto(link1.getTravelTime() - 18.8753138554997) < 0.81)
33
34     auto.testAuto(link2.getTravelTime() - 12.378) < 0.81)
35
36     link1.setFlow(8)
37     link2.setFlow(2512)

```

Here is the expected output if the function is implemented correctly:



```

10.102842197719283
12.378
10.0
19.341441772853184

*****
*   Autograde   *
*****
Testing Link.getTravelTime(): 2 / 4 (50%)

Total: 2 / 4 (50%)

*****
* End Autograde *
*****

```

**A note on testing.** The `autograde()` functions of each `Exercise.py` file are merely provided to check correctness. If your code is not correct, they will not indicate what the error is. This is to encourage good testing practice. In the `test()` function of each `Exercise.py` file, some code is provided which constructs `Links`, `Nodes`, or anything else relevant to the exercise. You can use this space to test the output of the function you wrote for the exercise and compare it to what you calculate by hand to be the correct answer. Once you believe your code is correct, use the `autograde()` function to verify correctness.

## 2.2 Iterating through links

Read the tutorials on [defining functions](#). When working with functions within classes, remember to use the “`self`” reference to access instance variables.

**Exercise 2(a)** Open `Link.py`. Implement the following:

- `getFlow()` function: returns the flow of the link

- `getCapacity()` function: returns the capacity of the link

*Hint:* the link flow is  $x_{ij}$  and the link capacity is  $C_{ij}$ . You already have variables for these values. It is good programming practice to separate the variables from other parts of the code through accessor functions.

Read the tutorials on [booleans](#). Then, read the tutorial on using `boolean` values to control the program flow through [if, elif, and else statements](#).

Now we need to introduce the first data structure, a list. A list is simply an ordered list of elements with a fixed size. Read the [tutorial on lists](#). When working with lists, it is helpful to use loops. Read the tutorials on [looping through lists](#). Loops are an essential element of programming. Read the tutorials on [for loops](#) and [while loops](#), and [breaking a loop](#).

We will be working with strings in the next exercise, so read the [tutorial on strings](#).

**Exercise 2(b)** Open `Exercise2.py`. Your task is to implement the `findCongestedLinks()` function in `Exercise2.py`, which outputs a string containing some information about each link. The list of links is passed as a function parameter. For each link in the list, first add “link” to the string and then the index of the link starting at 1, i.e. “1”, “2”, etc. Then add the link travel time, and finally add “yes” if  $x_{ij}/C_{ij} > 1$ , or “no” if  $x_{ij}/C_{ij} \leq 1$ . After each link, add a newline to the string by adding “\n”.

You may need to [cast numbers as a string](#).

Return the completed string as the output of the `findCongestedLinks()` function. Your output should look something like this:

```
link 1 10.171386840006189 yes
link 2 7.69733539223671 no
...
```

After completing Exercises 2(a) and 2(b), your code should pass the `autograde()` function of `Exercise2.py`.

## 3 Object-oriented programming

### 3.1 Network structure

You have already been working with the `Link` class to represent links in the network. It is time to learn enough about object-oriented programming to represent the entire network  $\mathcal{G}$ . Read the tutorial on [classes](#). You have already worked with the `getTravelTime()`, `getCapacity()`, and `getFlow()` functions of the `Link` class. Note that when a new instance of a class is created, the constructor function `__init__()` is automatically invoked.

When creating a new integer or float variable, you can set the initial value to 0 or 0.0. When creating a new object, e.g. an instance of `Link`, the default or empty value is `None`.

**Exercise 3(a)** Open `Node.py`. You will notice that a `Node` class has already been created for you. Open `Link.py`. Implement the following:

- `getStart()` function: returns the start node of the link.
- `getEnd()` function: returns the end node of the link.

In terms of the model, link  $(i, j)$  has start node  $i$  and end node  $j$ . These are already stored as instance variables in the `Link` class.

Open `Link.py`. You will notice that the first function `__init__()` is a constructor which stores the passed link parameters `start`, `end`, `t_ff`, `C`, `alpha`, and `beta` in the instance variables.

**Exercise 3(b)** Open `Node.py`. Implement the function `getId()` of the `Node` class. Each node has an id — an integer that is used to identify the node. Ids start at 1 and increment upwards. When a node is constructed, the id is passed as a parameter into the constructor for the `Node` class. The constructor parameter is a single `int` representing the id of the node. Therefore, the constructor looks like `__init__(self, id)`.

Also implement the constructor for the `Node` class. To do so, you may need to add instance variables to the `Node` class, e.g. an instance variable storing the id of the node.

**Exercise 3(c)** When you print an instance of a class, it will by default call the `__str__()` method of that class. Unless you implement it, the output will be a memory reference that is usually not useful.

- Implement the `__str__()` function of the `Node` class to return a string containing the id of the node.
- Implement the `__str__()` function of the `Link` class to return the string “ $(i, j)$ ” where  $i$  and  $j$  are the ids of the start and end nodes. For instance, a link from node 1 to node 2 should have a `__str__()` output of “(1, 2)”.

**Exercise 3(d)**

- Open `Node.py`. Implement the `getOutgoing()` function of the `Node` class. It returns a list of links that are outgoing from the given node. For node  $i$ , the list of outgoing links contains all links  $(i, j)$  that start at  $i$ .

*Hint:* you will need to create a new instance variable in the `Node` class to store that list of outgoing links  $\Gamma_i^+$  — do so in the constructor of the `Node` class.

- Implement the `addOutgoingLink(ij)` function in the `Node` class which adds `Link ij` to the list that you just created.

Then call this function in the constructor of the `Link` class so that every time a `Link` from  $i$  to  $j$  is created, it is added to the list of outgoing links of  $i$ .

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



## 3.2 Inheritance

Our next step is to create a representation of the demand  $d_{rs}$ . To do so, we will create a new **Zone** class that is a special type of **Node**: the **Zone**  $r$  stores the demand  $d_{rs}$ . Read the tutorials on [inheritance](#). Open `Zone.py`. The **Zone** class extends the **Node** class, which is indicated by the definition “`class Zone(Node.Node)`”.

For the next exercise, we will be implementing the `getDemand(dest)` function of the **Zone** class, which returns the demand  $d_{rs}$  from node  $r$  (the **Zone** being referenced) to a destination node **dest**.

When we read the file, we will store demand in the **Zone** instance of  $r$  by calling the `addDemand(s, d)` function to add  $d$  demand from  $r$  to  $s$ . To assist in this implementation, learn about [maps](#).

**Exercise 4(a)** Open `Zone.py`. Implement the constructor of the **Zone** class. You can call methods of the parent class using the `super()` function.

### Exercise 4(b)

- Open `Zone.py`. Implement the `addDemand(dest, demand)` function of the **Zone** class. This function is called on zone  $r$  to increase  $d_{rs}$  for a parameter  $s$ . This function must store the demand added for later reference by the `getDemand(dest)` method.
- Implement the `getDemand(s)` function, which returns the total demand from  $r$  to  $s$ .

*Hint:* Create a dictionary instance variable in the **Zone** class to store the demand.

**Exercise 4(c)** Open `Zone.py`. The productions of a zone  $P_r$  is defined as  $P_r = \sum_{s \in \mathcal{Z}} d_{rs}$ . Implement the `getProductions()` function of the **Zone** class, which returns the total productions of the zone. *Hint:* Iterate through all stored demand.

**Exercise 4(d)** Some zones are not through nodes, meaning that they can be used as destinations but not as intermediate nodes for travel. The function `isThruNode()` of class **Node** returns a boolean indicating whether a **Node** is a through node.

In the **Node** class, the function always returns **True**. Some **Zones** may not be a through node, meaning that the function should return **False** for them. In a later exercise, we will identify which nodes are through nodes.

Read the tutorial on [inheritance](#) to learn about overloading functions.

- Open `Zone.py`. Implement the `setThruNode()` function: it takes a boolean parameter indicating whether this zone is a through node. You will need to store it in an instance variable to implement the next function.
- Implement the `isThruNode()` function. It returns a boolean indicating whether the zone is a through node.

After completing Exercises 4(a)–4(d), your code should pass the `autograde()` method of `Exercise4.java`.

### 3.3 Reading network from files

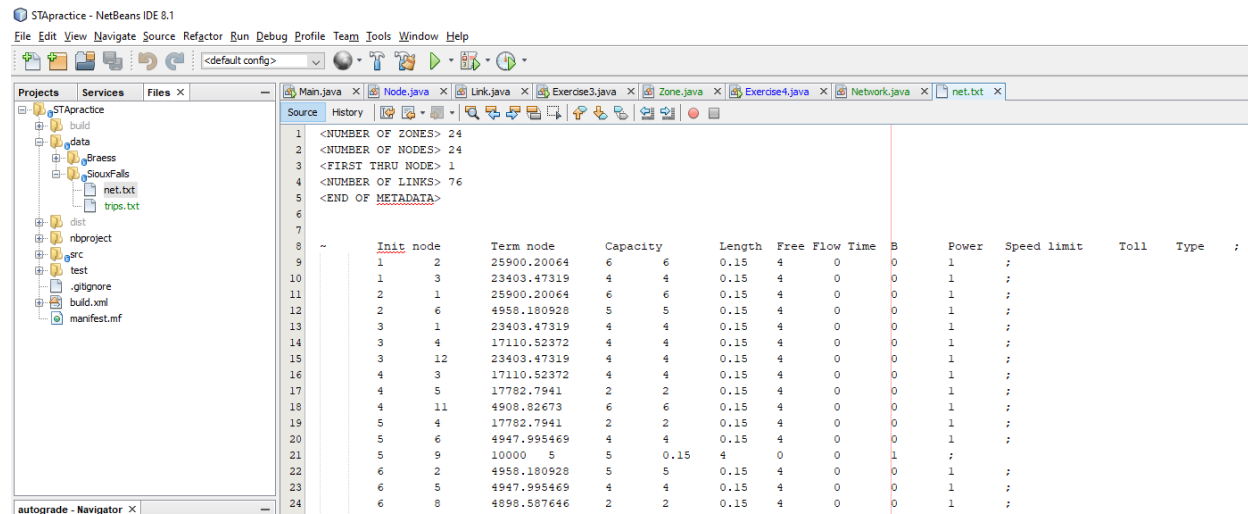
Open `Network.py`. It contains the `Network` class, which represents  $\mathcal{G}$  in the network definition  $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ . Therefore, it stores the nodes and links of the network, and includes some functions for network calculations. It has been partially implemented for you. There are instance variable lists of `Nodes`, `Links`, and `Zones`, which represent the sets  $\mathcal{N}$ ,  $\mathcal{A}$ , and  $\mathcal{Z}$  of the network. There are also accessor methods for each of these lists.

The next step is to populate these sets with network data. Thus far, we have been creating specific instances of `Nodes` and `Links` in the `Exercise.py` files. To keep our code more general, we want to keep the problem-specific data in data files rather than in the code. Fortunately, data for many networks is available on [Ben Stabler's Github account](#).

Before we discuss the data format, we need to learn how to read from a file. Read the tutorial on [reading from a file](#). The functions `readline()` and `readlines()` methods may be useful here also.

We will need to extract individual numbers from the file. I suggest using the `split()` function. Occasionally, when you read a line, you will end up with extra whitespace at the end, e.g. in the string “extra space”. You can remove that using the `strip()` function. You may come across a number that is given as a string, e.g. the string “26.2”, but you want to convert it to integer or float type. Use the `casting functions`.

In this project, the network data is contained within the folder “data/[network name]/”. Each network is specified by two text files, “net.txt” and “trips.txt”. The constructor `Network(name)` constructs the `Network` by calling the `readNetwork(file)` and `readTrips(file)` functions for the given network name. The first file, “net.txt”, defines the links and their characteristics. An example is shown below:



```
1 <NUMBER OF ZONES> 24
2 <NUMBER OF NODES> 24
3 <FIRST THRU NODE> 1
4 <NUMBER OF LINKS> 76
5 <END OF METADATA>
6
7
8 ~ Init node Term node Capacity Length Free Flow Time B Power Speed limit Toll Type ;
9 1 2 25900.20064 6 6 0.15 4 0 0 1 ;
10 1 3 23403.47319 4 4 0.15 4 0 0 1 ;
11 2 1 25900.20064 6 6 0.15 4 0 0 1 ;
12 2 6 4958.180928 5 5 0.15 4 0 0 1 ;
13 3 1 23403.47319 4 4 0.15 4 0 0 1 ;
14 3 4 17110.52372 4 4 0.15 4 0 0 1 ;
15 3 12 23403.47319 4 4 0.15 4 0 0 1 ;
16 4 3 17110.52372 4 4 0.15 4 0 0 1 ;
17 4 5 17782.7941 2 2 0.15 4 0 0 1 ;
18 4 11 4908.82673 6 6 0.15 4 0 0 1 ;
19 5 4 17782.7941 2 2 0.15 4 0 0 1 ;
20 5 6 4947.995469 4 4 0.15 4 0 0 1 ;
21 5 9 10000 5 5 0.15 4 0 0 1 ;
22 6 2 4958.180928 5 5 0.15 4 0 0 1 ;
23 6 5 4947.995469 4 4 0.15 4 0 0 1 ;
24 6 8 4898.587646 2 2 0.15 4 0 0 1 ;
```

The first section contains the metadata, which specifies the size of the sets  $\mathcal{N}$ ,  $\mathcal{A}$ , and  $\mathcal{Z}$ . This section is ended by the line “`<END OF METADATA>`”. Nodes are labeled by the numbers  $1 \dots |\mathcal{N}|$  where  $|\mathcal{N}|$  is specified in the metadata.

**Exercise 5(a)** Open `Network.py`. Add code in the `readNetwork()` function to read the metadata to obtain the numbers of nodes, zones, and links.

*Hint:* loop until the line <END OF METADATA> is reached. If an intermediate line contains the text <NUMBER OF NODES>, then use that number to instantiate the **nodes** array. Repeat for the links and zones.

### Exercise 5(b)

- Open Network.py. Zones are labeled 1 through  $|\mathcal{Z}|$ . Populate the instance variable list **zones** in the **Network** class by constructing a new instance of **Zone** in the **readNetwork()** function and store them in the list **zones**.

*Hint:* the number of nodes in the metadata specifies the number of zones to construct.

- Populate the instance variable list **nodes** in the **Network** class. Nodes are labeled 1 through  $|\mathcal{N}|$ , which includes some nodes that are also zones. Construct new nodes in the **readNetwork()** function and add them to the list **nodes** as needed.

Do not construct new instances of **Node** for zones that you already constructed. Use the instance of **Zone** that already exists in the array **zones**.

**Exercise 5(c)** After the header line, each line of data contains the parameters for one link in a specific order: start node, end node, capacity, length, free flow time,  $\alpha_{ij}$ ,  $\beta_{ij}$ , speed limit, toll, and type. “B” refers to  $\alpha_{ij}$  and “power” refers to  $\beta_{ij}$ . Some of these are not used in this tutorial.

In the **readNetwork()** function of the **Network** class, read the data and use it to construct a new **Link** instance for each link. Store those instances in the list **links**. *Hint:* the number of links in the metadata specifies the number of lines of data.

The second file is “trips.txt”. The metadata here can be ignored. For each zone  $r$ , the keyword **Origin** defines the start of the demand array  $d_{rs}$  for each zone  $s$ . That demand is specified over the next several lines.

```

1 <NUMBER OF ZONES> 24
2 <TOTAL OD FLOW> 360600.0
3 <END OF METADATA>
4
5
6 Origin 1
7   1 :    0.0;    2 :   100.0;    3 :   100.0;    4 :   500.0;    5 :   200.0;
8   6 :   300.0;    7 :   500.0;    8 :   800.0;    9 :   500.0;   10 :  1300.0;
9  11 :   500.0;   12 :   200.0;   13 :   500.0;   14 :   300.0;   15 :   500.0;
10 16 :   500.0;   17 :   400.0;   18 :   100.0;   19 :   300.0;   20 :   300.0;
11 21 :   100.0;   22 :   400.0;   23 :   300.0;   24 :   100.0;
12
13 Origin 2
14   1 :   100.0;    2 :    0.0;    3 :   100.0;    4 :   200.0;    5 :   100.0;
15   6 :   400.0;    7 :   200.0;    8 :   400.0;    9 :   200.0;   10 :   600.0;
16  11 :   200.0;   12 :   100.0;   13 :   300.0;   14 :   100.0;   15 :   100.0;
17  16 :   400.0;   17 :   200.0;   18 :    0.0;   19 :   100.0;   20 :   100.0;
18  21 :    0.0;   22 :   100.0;   23 :    0.0;   24 :    0.0;
19

```

**Exercise 5(d)** Open `Network.py`. In the `readTrips()` function, read the origin-destination trip matrix, and store it in the `Zone` instances using the `addDemand()` function of class `Zone`.

*Hint:* Iterate through each line to the end of the metadata using a `while` loop until you reach the text `<END OF METADATA>`. Then look for origins denoted by the `Origin` keyword. Then look for destination node ids until you reach the next `Origin` keyword. Use the `split()` function to obtain individual data, e.g. “Origin”, zone ids, demand numbers, etc.

**Exercise 5(e)** For testing or data analysis, you may find it helpful to find the `Node` associated with a given id, or the `Link` between two `Nodes`.

- Open `Network.java`. Implement the `findNode()` function, which returns the node with the given id, or `None` if none exists.
- Implement the `findLink()` function, which returns the link between the 2 given nodes, or `None` if none exists.

*Hint:* You have a list of all `Nodes` available in the `Network` class, and a list of all outgoing links from a given `Node`.

After completing Exercises 5(a)–5(e), your code should pass the `autograde()` method of `Exercise5.py`.

## 4 Data structures and algorithms

### 4.1 Dijkstra’s algorithm

We are now ready to implement a shortest path algorithm, which will be used in the method of successive averages. We will implement the well-known Dijkstra’s algorithm, which finds the one-to-all shortest path. For more information on Dijkstra’s, see [Transportation Network Analysis](#). We need to define two additional variables. Let  $c_n \in \mathbb{R}_+$  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:

```

1: procedure DIJKSTRA’S( $r$ )
2:   for  $n \in \mathcal{N}$  do                                     ▷ Initialization
3:      $c_n \leftarrow \infty$ 
4:      $p_n \leftarrow \emptyset$ 
5:   end for
6:    $c_r \leftarrow 0$ 
7:    $Q \leftarrow \{r\}$ 

8:   while  $Q \neq \emptyset$  do                               ▷ Main loop
9:      $u \leftarrow \arg \min_{n \in Q} \{c_n\}$ 
10:     $Q \leftarrow Q / \{u\}$ 
11:    for  $(u, v) \in \mathcal{A}$  do
12:      if  $c_u + t_{uv} < c_v$  then                         ▷ Is this a shorter path to  $v$ ?
13:         $c_v \leftarrow c_u + t_{uv}$                          ▷ 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  $c_n \leftarrow \infty$ . The operator  $\leftarrow$  is used to indicate that  $c_n$  is assigned the value  $\infty$ , 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  $\arg \min_{n \in Q} \{c_n\}$  could involve looping through all elements of  $Q$  to find the  $n$  with the smallest value of  $c_n$ . 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  $c_n$  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.
- `size()`: returns the size of the list.

In addition, `Path` 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  $T^\pi$ .
- `addHstar()` will be used later.

### Exercise 6(a)

- Open Path.py. Implement the `getSource()` function: return the starting node of the path.  
*Hint.* You have a list of all links in the path.
- Implement the `getDest()` function: return the last node of the path.
- Implement the `isConnected()` function, which checks whether the list of links is a connected path.
- Implement the `getTravelTime()` function, which returns  $T^\pi = \sum_{(i,j) \in \pi} t_{ij}(x_{ij})$ .

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

**Exercise 6(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 before proceeding further.

**Exercise 6(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.

```
1: procedure TRACE( $r, s$ )
2:    $n \leftarrow s$ 
3:    $\pi \leftarrow \emptyset$ 
4:   while  $n \neq r$  do
5:      $\pi \leftarrow \pi \cup \{(p_n, n)\}$ 
6:      $n \leftarrow p_n$ 
7:   end while
8: end procedure
```

**Exercise 6(d)** Open Network.py. Implement the `trace()` function in the `Network` class. Remember 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 6(a)–6(d), your code should pass the `autograde()` method of `Exercise6.py`.

## 4.2 Network statistics

Before implementing the method of successive averages, there are some network statistics that will be used in the implementation. These are the total system travel time,  $TSTT$ , the shortest path travel time,  $SPTT$ , and the average excess cost,  $AEC$ . These are defined mathematically as follows:

$$TSTT = \sum_{(i,j) \in \mathcal{A}} x_{ij} t_{ij}(x_{ij}) \quad (5)$$

$$SPTT = \sum_{(r,s) \in \mathcal{Z}^2} \mu_{rs} d_{rs} \quad (6)$$

$$AEC = \frac{TSTT - SPTT}{\sum_{(r,s) \in \mathcal{Z}^2} d_{rs}} \quad (7)$$

### Exercise 7

- Open Network.py. Implement the `getTSTT()` function: return the total system travel time.
- Implement the `getSPTT()` function: return the total system travel time if all demand is on the shortest path.
- Implement the `getAEC()` function: return the average excess cost.

After completing them, your code should pass the `autograde()` function of Exercise7.py.

## 4.3 Method of successive averages

The method of successive averages is a simple algorithm for solving user equilibrium. Each iteration, it constructs an all-or-nothing flow assignment  $\mathbf{x}^*$  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  $\lambda$ . 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 excess cost 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:      $x_{ij}^* \leftarrow 0$ 
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:          $\pi_{rs}^* \leftarrow \text{TRACE}(r, s)$ 
10:        for  $(i, j) \in \pi_{rs}^*$  do ▷ Update all-or-nothing flow assignment
11:           $x_{ij}^* \leftarrow x_{ij}^* + d_{rs}$ 

```

```

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  $\mathbf{x}^*$ 
17:          $x_{ij} \leftarrow (1 - \lambda)x_{ij} + \lambda x_{ij}^*$ 
18:          $x_{ij}^* \leftarrow 0$ 
19:     end for
20:     PRINT(AEC)                                            ▷ Track convergence
21: end for
22: end procedure

```

### Exercise 8(a)

- Open Link.py. Create a new instance variable to store  $x_{ij}^*$  in the **Link** class.
- In the **Link** class, implement the `addXstar()` function which adds the specified flow to the  $x_{ij}^*$  variable. It will be used to implement line 11.
- In the **Path** class, implement the `addHstar()` function which adds the specified flow to the  $x_{ij}^*$  variable of every link in the path. Use the `addXstar()` method of the **Link** class in your implementation.

### Exercise 8(b)

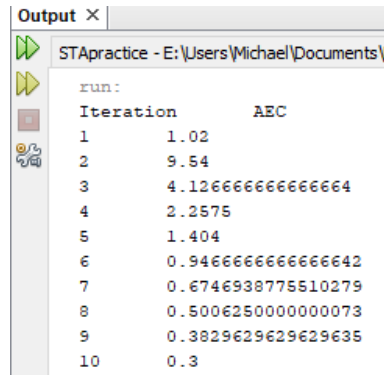
- Open Network.py. In the **Network** class, implement the `calculateStepsize()` function, which determines the value of  $\lambda$  in line 15.
- Open Link.py. Using this  $\lambda$ , in the **Link** class implement the `calculateNewX()` function, which takes as input  $\lambda$  and implements lines 17 and 18.
- Open Network.py again, and implement the `calculateNewX()` function of the **Network** class, which implements the loop in line 16.

**Exercise 8(c)** Open Network.py. Implement the `calculateAON()` function in the **Network** class, which is the loop in lines 6–14.

**Exercise 8(d)** In the method `msa(int)` of the **Network** class, implement the main loop (line 5 of the method of successive averages. Most of the work is already done through the previous exercises — `Network.calculateAON()`, `Network.calculateStepSize()`, `Link.calculateNewX()`, etc.

The output is a string containing the following: for each iteration, include a row with the iteration number and the average excess cost, as shown below:





Iteration	AEC
1	1.02
2	9.54
3	4.126666666666664
4	2.2575
5	1.404
6	0.9466666666666642
7	0.6746938775510279
8	0.5006250000000073
9	0.3829629629629635
10	0.3

After completing Exercises 8(a)–8(d), your code should pass the `autograde()` method of `Exercise8.java`.

## 5 Next steps

This is not the most efficient implementation of the method of successive averages. Now that you have a correct implementation, you may want to go back and improve the computational efficiency. In addition, the method of successive averages is far from the most efficient algorithm. The Frank-Wolfe algorithm can be implemented in this code fairly easily. You may also wish to try implementing gradient projection (Jayakrishnan et al., 1994) or Algorithm B (Dial, 2006).

## References

- R. B. Dial. A path-based user-equilibrium traffic assignment algorithm that obviates path storage and enumeration. *Transportation Research Part B: Methodological*, 40(10):917–936, 2006.
- R. Jayakrishnan, W. T. Tsai, J. N. Prashker, and S. Rajadhyaksha. A faster path-based algorithm for traffic assignment. 1994.