A tutorial for programming static traffic assignment in Java

Michael W. Levin

June 3, 2021

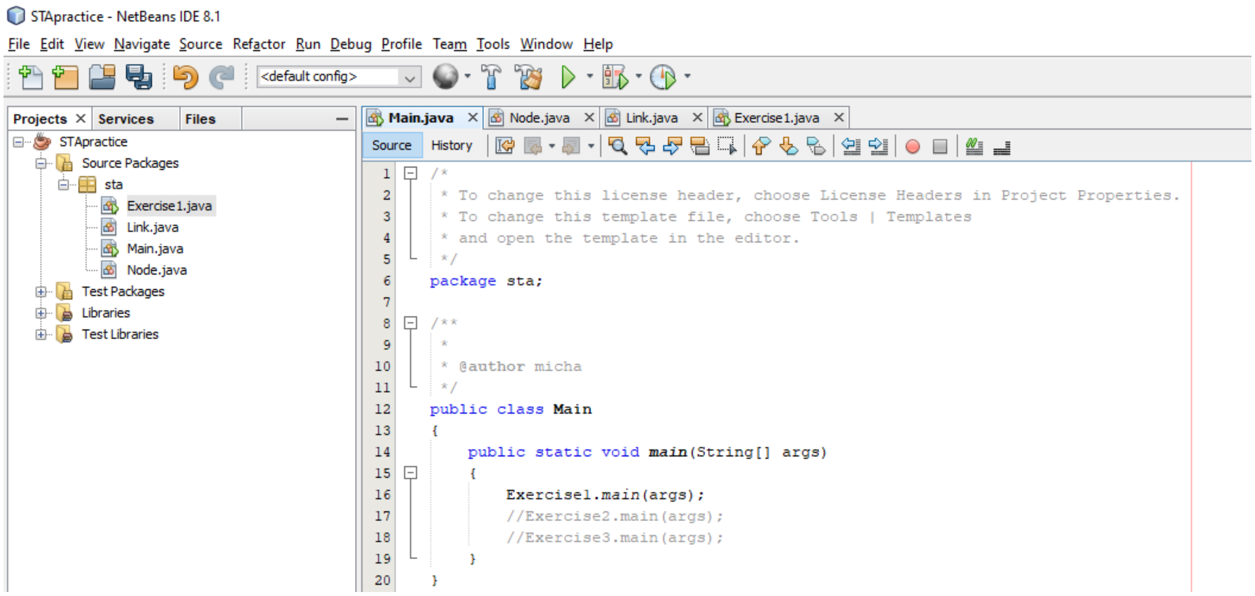
1 Introduction

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 Java programming language, which has potential for high-performance computing, is designed around object-oriented programming, and avoids low-level memory management.

1.1 Getting started

The existing code is provided as a Netbeans project. Download and install Netbeans and the Java Development Kit. Download a copy of this Git repository:
https://github.com/mwlevin/STApractice.git. Alternatively, you can clone it in Netbeans.

The repository contains a Netbeans project, which you may open directly in Netbeans. The `main()` method (which is executed when you run the program) is found in Main.java. Each of the exercises in this tutorial are contained within a separate file, e.g. Exercise1.java, Exercise2.java, etc. Each of these files has their own `main` method that can be executed. In Main.java, you will find calls to these methods commented out, i.e. `Exercise1.main(args);`. Uncomment them to run each exercise. These exercises are designed to be completed sequentially as they build on the code written previously. The autograde may not be able to check correctness if you complete them out of order.



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

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

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.

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 π consists of a set of links. Let Π be the set of all paths, and let $\Pi_{rs} \subseteq \Pi$ be the set of paths from r to s. Let h^π be the flow on path π, and let T^π be the travel time for path π. Let $\delta_{ij}^\pi \in \{0,1\}$ indicate whether path π includes link (i,j). Then T^π can be written as

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

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

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

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

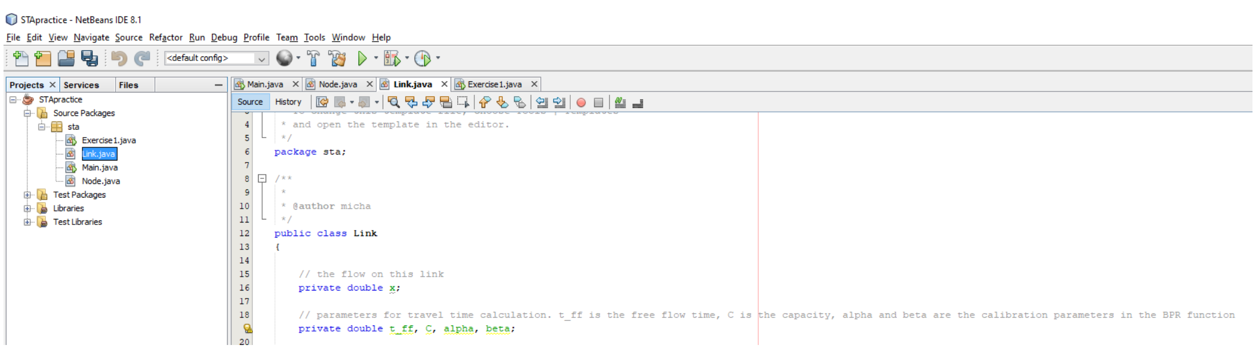
$$
\begin{aligned}
\min \quad & Z = \sum_{(i,j)\in\mathcal{A}} \int_0^{x_{ij}} t_{ij}(\omega)d\omega && && \text{(4a)}\\
\text{s.t.} \quad & x_{ij} = \sum_{\pi\in\Pi} \delta_{ij}^{\pi} h^{\pi} && \forall (i,j)\in\mathcal{A} && \text{(4b)}\\
& d_{rs} = \sum_{\pi\in\Pi_{rs}} h^{\pi} && \forall (r,s)\in\mathcal{Z}^2 && \text{(4c)}\\
& h^{\pi} \geq 0 && \forall \pi\in\Pi && \text{(4d)}
\end{aligned}
$$

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 Java syntax and comments. Read the tutorials on variables, data types, and type casting. Open Link.java in Netbeans. You will notice that the code first defines a `public class Link`, which is intended to 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 Java. You will see some variables `x`, `t_ff`, `C`, `alpha`, and `beta` defined here. For now, it is sufficient to know that these variables are available for use anywhere within the `Link` class. These variables correspond to the model variables x_{ij}, t_{ij}^{ff}, C_{ij}, α_{ij}, and β_{ij} for the specific link (i,j) being represented.



Read the tutorials on operators and the Math package. Here is a list of all Math methods.

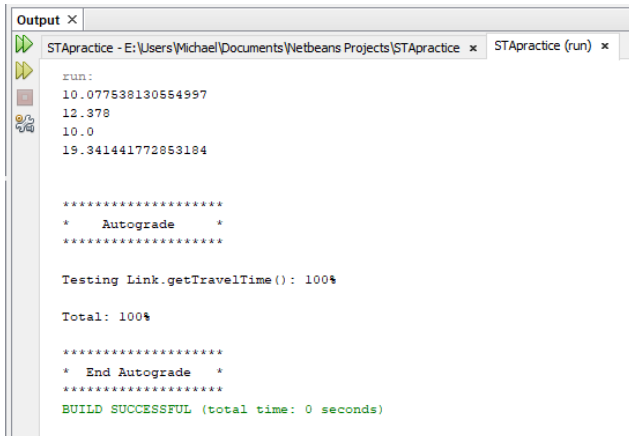
Exercise 1 Your first task is to implement the calculation of the link travel time $t_{ij}(x_{ij})$ using the BPR function of equation (1). Assume that the values of x_{ij}, t_{ij}^{ff}, C_{ij}, α_{ij}, and β_{ij} are already given. Within the `Link` class, find the method `getTravelTime()`. It defines a `double t_ij` and sets the value to 0. You need to calculate the correct value of $t_{ij}(x_{ij})$ and assign it to variable `t_ij`.

```
/* **********
Exercise 1
********** */
public double getTravelTime()
{
    // fill this in
    double t_ij = 0;


    return t_ij;
}
}
```

Open Main.java and ensure that it will run `Exercise1.main(args);`. Open Exercise1.java. 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()` method 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, `main` calls the `autograde()` method, which runs an automated test of your answers.

```
public class Exercise1
{
    public static void main(String[] args)
    {
        Link link1 = new Link(null, null, 10, 2580, 0.15, 4);
        Link link2 = new Link(null, null, 12, 1900, 0.35, 2);

        link1.setFlow(1230.2);
        System.out.println(link1.getTravelTime());


        link2.setFlow(570);
        System.out.println(link2.getTravelTime());

        link1.setFlow(0);
        link2.setFlow(2512);

        System.out.println(link1.getTravelTime());
        System.out.println(link2.getTravelTime());


        autograde();
    }
```

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



A note on testing. The `autograde()` methods of each Exercise.java 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 `main()` method of each Exercise.java file, some code is provided which constructs `Link`s, `Node`s, or anything else relevant to the exercise. You can use this space to test the output of the methods 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()` method to verify correctness.

2.2 Iterating through links

Read the tutorials on defining methods and method parameters. When working with methods, it is important to be aware of the scope of variables.

Exercise 2(a) Open Link.java. Implement the `getFlow()` and `getCapacity()` methods. *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 methods.

Read the tutorials on booleans. Then, read the tutorial on using `boolean` values to control the program flow through `if` and `else` statements. You may also find it interesting to read about `switch` logic, but `switch` statements can also be accomplished by `if` and `else if` statements.

Now we need to introduce the first data structure, arrays. An array is simply an ordered list of elements with a fixed size. Read the tutorial on arrays. When working with arrays, it is helpful to use loops. Read the tutorials on `while` loops, `for` loops, and loop control.

Exercise 2(b) Open Exercise2.java. Your task is to implement the `findCongestedLinks()` method, which prints some information about each link. The array of links is passed as a method parameter. For each link in the array, first print "link" and the link number, i.e. "1", "2", etc. Then print the link travel time, and finally print "yes" if $x_{ij}/C_{ij} > 1$, or "no" if $x_{ij}/C_{ij} \leq 1$. Use `System.out.print()` and `System.out.println()` to print to the console. 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()` method of Exercise2.java.

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 tutorials on object-oriented programming, classes, instance variables, and class methods. You have already worked with the `getTravelTime()`, `getCapacity()`, and `getFlow()` methods of the `Link` class. Read the tutorial on different types of class methods. When a new instance of a class is created, the constructor method is automatically invoked. Read the tutorials on constructor methods.

Exercise 3(a) Open Node.java. You will notice that a `Node` class has already been created for you. Open Link.java. Implement the `getStart()` and `getEnd()` methods of the `Link` class, which return the start and end nodes. 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.java. You will notice that the first method is a constructor which stores the passed link parameters `start`, `end`, `t_ff`, `C`, `alpha`, and `beta` in the instance variables.

Exercise 3(b) Implement the method `getId()` of the `Node` class. 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 `public Node(int id)`. Also implement the constructor for the `Node` class. To do so, you may need to add instance variables to the `Node` class.

Exercise 3(c) Text is represented as `Strings` in Java. Read the tutorial on `Strings`. When you print an instance of a class, it will by default call the `toString()` method of that class. Unless you implement it, the output will be a memory reference that is usually not useful. Implement the `toString()` methods of the `Node` class to return the id of the node. Also implement the `toString()` method of the `Link` class to return "(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 `toString()` output of "`(1, 2)`".

It is now helpful for us to learn about dynamic arrays. Read the tutorial on the Java `ArrayList` library. Later on, we will need to obtain Γ_i^+: the set of links starting at i. This is defined as the `getOutgoing()` method of the `Node` class, which returns an `ArrayList` of all outgoing `Link`s.

Exercise 3(d) Implement the `getOutgoing()` method of the `Node` class. *Hint:* you will need to create a new instance variable `ArrayList` in the `Node` class to store Γ_i^+. You will need to instantiate this new `ArrayList` — do so in the constructor of the `Node` class. Create a new method in the `Node` class to add `Link`s to this `ArrayList`. Then call this method in the constructor of the `Link` class. The keyword `this`, which refers to the current instance of a class, is useful here.

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

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 and polymorphism. Open Zone.java. The `Zone` class extends the `Node` class. For the next exercise, we will be implementing the `getDemand(Node)` method of the `Zone` class, which returns the demand d_{rs} from node r (the `Zone` being referenced) to a destination node. Demand is added via the `addDemand(Node, double)` method. To assist in this implementation, learn about `HashMaps`. The `hashCode` method, which is needed to use `Node`s as a key for `HashMap`s, has already been implemented for you.

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

Exercise 4(b) Implement the `addDemand(Node, double)` method of the `Zone` class. This method must store the demand added for later reference by the `getDemand(Node)` method. Implement the `getDemand(Node)` method. *Hint:* Create a `HashMap` instance variable in the `Zone` class to store demand. Instantiate this `HashMap` in the constructor of the `Zone` class.

Exercise 4(c) The productions of a zone P_r is defined as $P_r = \sum_{s \in \mathcal{Z}} d_{rs}$. Implement the `getProductions()` method of the `Zone` class. *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 `boolean` method `isThruNode()` of class `Node` indicates whether a `Node` is a through node. In the `Node` class, the method always returns `true`. Some `Zone`s may return `false`. Read the tutorial on method overloading. Implement methods `setThruNode(boolean)` and `isThruNode()` of the `Zone` class. `Zone`s are through nodes by default, but that parameter can be changed by calling `setThruNode(false)`.

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

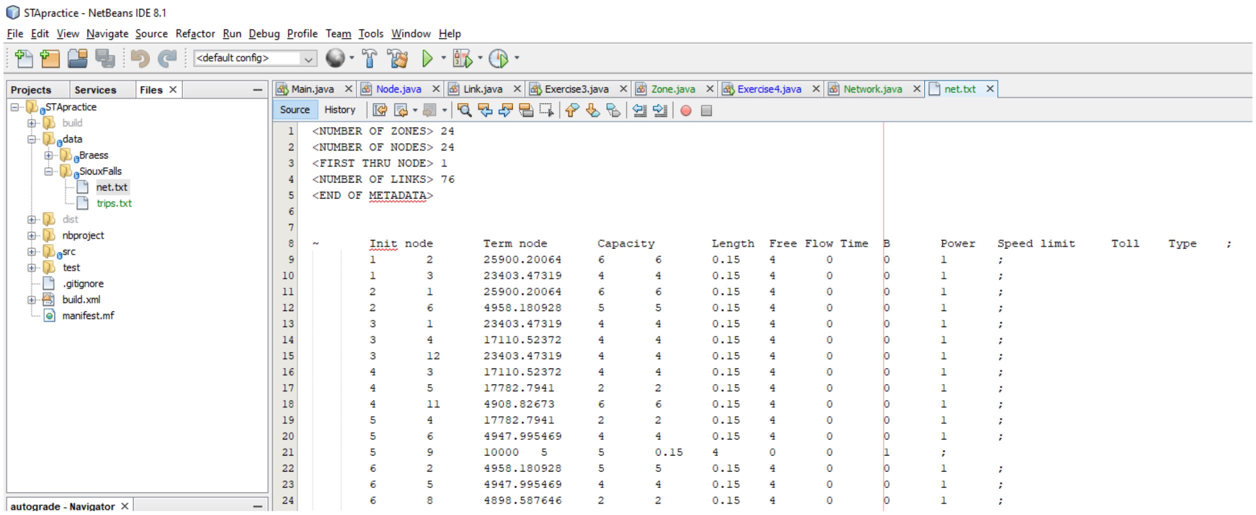
3.3 Reading network from files

Open Network.java. It contains the `Network` class, which has been partially implemented for you. There are instance variable arrays of `Node`s, `Link`s, and `Zone`s, which represent the sets $\mathcal{N}$, $\mathcal{A}$, and $\mathcal{Z}$ of the network. There are also accessor methods for each of these arrays. These arrays need to be instantiated to the correct size.

The next step is to populate these sets with network data. Thus far, we have been creating specific instances of `Node`s and `Link`s in the Exercise.java files. To keep our code 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 the `Scanner` class, which is an effective way of reading different data from a file in text format. We need to direct the `Scanner` to use a file as an input source. Read the tutorials on the `File` class and reading from a file.

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(String)` constructs the `Network` by creating the appropriate `File` references and calling the `readNetwork(File)` and `readTrips(File)` methods for the given network name. The first file, "net.txt", defines the links and their characteristics. An example is shown below:



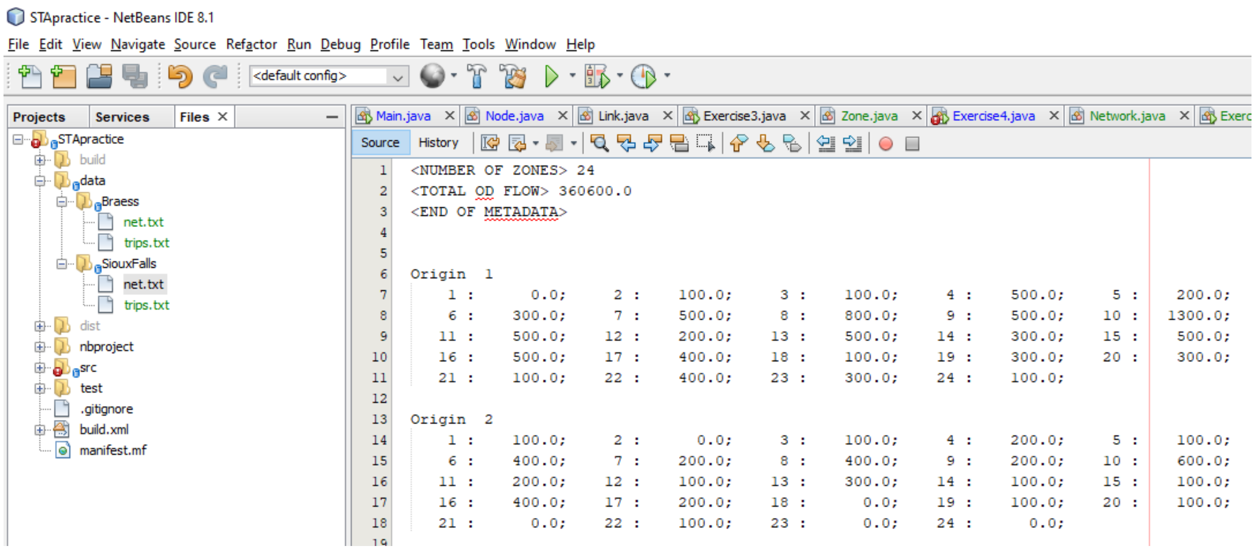
The first section contains the metadata, which specifies the size of the sets $\mathcal{N}$, $\mathcal{A}$, and $\mathcal{Z}$. This section is ended by the line `<END OF METADATA>`. Nodes are labeled by the numbers $1 \ldots |\mathcal{N}|$ where $|\mathcal{N}|$ is specified in the metadata.

Exercise 5(a) Read the metadata to instantiate the instance variables `nodes`, `links`, and `zones` of the `Network` class with the correct size. *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`. The String methods `substring()`, `indexOf()`, and `trim()` are useful here. Use the method `Integer.parseInt(String)` to convert a `String` representing a number into an `int`. The equivalent method for `doubles` is `Double.parseDouble(String)`.

Exercise 5(b) Zones are labeled 1 through $|\mathcal{Z}|$. Construct these in the `readNetwork(File)` method of the `Network` class and store them in the array `zones`. *Hint:* the number of nodes in the metadata specifies the number of zones to construct. Nodes are labeled 1 through $|\mathcal{N}|$. You can similarly construct them now, but notice that the first $|\mathcal{Z}|$ nodes are also zones. Do not construct new instances of `Node` for these zones; use the instance of `Zone` that already exists.

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, α_{ij}, β_{ij}, speed limit, toll, and type. "B" refers to α_{ij} and "power" refers to β_{ij}. Some of these are not used in this tutorial. In the `readNetwork()` method of the `Network` class, read the data using `Scanner` methods to construct a `Link` instance for each link, and store it in the array `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.



Exercise 5(d) In the `readTrips(File)` method, read the origin-destination trip matrix, and store it in the `Zone` instances using the `addDemand(Node, double)` method of class `Zone`. *Hint:* Scan to the end of the metadata using a `while` loop until you reach the text `<END OF METADATA>`. Scan for origins the `Origin` keyword. Then scan for destinations as integers until you reach the next `Origin` keyword. Use `Scanner.next()` instead of looking for new lines.

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`. Implement the `findNode(int)` and `findLink(Node, Node)` methods of the `Network` class. *Hint:* You have an array 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.java.

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 ∞. p_n is assigned the value $\emptyset$, or `null` in Java, 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. 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 method `getOutgoing()` of the `Node` class which you implemented previously will be useful here. In line 12, notice that while c_u and c_v are variables, t_{uv} is a method 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 the `ArrayList`). 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 `Link`s. To provide practice with abstraction, I have defined a `Path` class in Path.java. A `Path` extends an `ArrayList<Link>`, which inherits the `get()`, `size()`, and `add()` methods of `ArrayList`. In addition, `Path` defines five additional methods. `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 start and end nodes of the path. `getTravelTime()` calculates T^{π}. The last method, `addHstar(double)`, will be used later.

Exercise 6(a) Open Path.java. Implement the `getSource()`, `getDest()`, `isConnected()`, and `getTravelTime()` methods of the `Path` class.

To implement Dijkstra's, we need two additional variables c_n and p_n. These are available as the instance variables `cost` and `predecessor` of the `Node` class. Notice that unlike other instance variables, these have been declared using the `protected` keyword. Consequently, these variables can be accessed and modified directly from other classes without going through methods. Review the different types of access modifiers.

Exercise 6(b) Open Network.java and navigate to the `dijkstras(Node)` method. Implement the initialization (lines 2–7) of Dijkstra's algorithm. Use `Double.MAX_VALUE` in lieu of ∞. It may be convenient to use a `HashSet` to implement Q. You may wish to test the correctness of the initialization before proceeding further.

Exercise 6(c) In Network.java, implement the main loop of Dijkstra's algorithm (lines 8–18) in the `dijkstras(Node)` method.

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 algorithmic 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 6(d) Implement the `trace(Node, Node)` method 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()` method of the `Path` class.

After completing Exercises 6(a)–6(d), your code should pass the `autograde()` method of Exercise6.java.

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:

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

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

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

Exercise 7 Implement the `getTSTT()`, `getSPTT()`, `getTotalTrips()`, and `getAEC()` methods of the `Network` class. After completing them, your code should pass the `autograde()` method of Exercise7.java.

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 $\mathbf{x}^\star$ 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 excess cost each iteration. The algorithm is specified below in pseudocode:

procedure Method of successive averages(I)
  for $(i,j) \in \mathcal{A}$ **do** ▷ Initialization
    $x^\star_{ij} \leftarrow 0$
  end for
  for *iteration* $\leftarrow 1$ to I **do**
    for $r \in \mathcal{Z}$ **do**
      Dijkstra's(r) ▷ Find shortest paths from r to s
      for $s \in \mathcal{Z}$ **do**

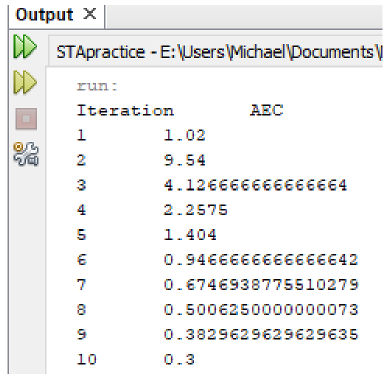
9: $\pi^\star_{rs} \leftarrow \textsc{trace}(r,\ s)$
10: **for** $(i,j) \in \pi^\star_{rs}$ **do** ▷ Update all-or-nothing flow assignment
11: $x^\star_{ij} \leftarrow x^\star_{ij} + d_{rs}$
12: **end for**
13: **end for**
14: **end for**
15: $\lambda \leftarrow \frac{1}{iteration}$ ▷ Calculate step 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: $\textsc{Print}(AEC)$ ▷ Track convergence
21: **end for**
22: **end procedure**

Exercise 8(a) In the `Link` class, implement the `addXstar(double)` method which is used to implement line 11. The $x^\star_{ij}$ value will need to be stored in a new instance variable in the `Link` class. In the `Path` class, implement the `addHstar(double)` which adds the specified flow to the $x^\star_{ij}$ variable of every link in the path.

Exercise 8(b) In the `Network` class, implement the `calculateStepsize(int)` method, which determines the value of λ in line 15. Using this λ, in the `Link` class implement the `calculateNewX(double)` method, which takes as input λ and implements lines 17 and 18. Finally, implement the `calculateNewX(double)` method of the `Network` class, which implements the loop in line 16.

Exercise 8(c) Implement the `calculateAON()` method in the `Network` class, which implements 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. Each iteration, print out the iteration number and the average excess cost, as shown below:



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.