Lesson 5. The Mileage Running Problem

The problem

Professor May B. Wright needs to fly from Baltimore (BWI) to Los Angeles (LAX) to attend a conference. She thinks this would be the perfect opportunity to accumulate some frequent flyer miles on American Airlines (AA), where she already has Platinum status.

Looking into flights on AA, she sees that every itinerary from BWI to LAX costs roughly the same. She has a full day to spare for travel, so she wants to know: which sequence of AA domestic flights starting at BWI and ending at LAX over the course of one day will allow her to accumulate the most miles?

- Yes, people actually do this. This is known as **mileage running**.
  - Apparently, this has become harder to do in recent years.
  - A recent article from the New York Times.
  - An older article from Wired.

Modeling the problem

- Suppose we have a database of every AA domestic flight on a given day.
- In particular, for each flight, we have:
  - the flight number
  - the origin airport
  - the destination airport
  - the departure time at the origin airport
  - the arrival time at the destination airport
  - the distance traveled in miles
- How can we formulate Professor Wright's problem as a shortest path problem?

pandas (the package, not the animals)

- In the same folder as this notebook, there is a file called `aa_domestic_flights.csv` with the database described above.
- `.csv` stands for **comma-separated values**.
- We can view `.csv` files in Excel - let's see what's in this file. *Cut to Excel...*
- How can we use this data in Python? With **pandas**.
- pandas is a Python package for data analysis.
  - It's especially useful for cleaning and manipulating datasets.
- pandas does a lot of stuff — here is the official documentation for pandas.

- In this lesson, we'll use pandas in a very basic way to set up the shortest path problem we formulated above.

- To install pandas, open a WinPython Command Prompt and type

```
pip install pandas
```

- `pip` might tell you that pandas is already installed. If not, it should go ahead and install it for you.
- To use pandas, we first need to import it, like this:

```
In [2]: import pandas as pd
```

- A pandas **DataFrame** is just a two-dimensional table, with rows and columns.
- We can use the `read_csv()` function in pandas to read `aa_domestic_flights.csv` into a DataFrame called `df`, like this:

```
In [3]: # Read csv file into a DataFrame
        # Designate departure and arrival time columns as dates
        df = pd.read_csv('aa_domestic_flights.csv', parse_dates=['DEP_TIME', 'ARR_TIME'])
```

- By default, `read_csv()` assumes the first row of the csv file contains the names of each column.
- The `parse_dates` argument tells `read_csv()` which columns correspond to dates, so that we can perform date-specific calculations on these columns later.
- Here is the official documentation for `read_csv()`.

- It's a good idea to take a quick look at the DataFrame `read_csv()` creates, just in case something went wrong.
- To examine the first 5 rows of a DataFrame, we can use the `.head()` method:

```
In [4]: # Print the first 5 rows of df
        df.head()
```

```
Out[4]:          FLIGHT ORIGIN DEST            DEP_TIME            ARR_TIME  DISTANCE
        0      1-BOS-JFK    BOS  JFK 2016-09-01 06:00:00 2016-09-01 07:15:00     187.0
        1     40-BOS-ORD    BOS  ORD 2016-09-01 19:12:00 2016-09-01 22:02:00     867.0
        2    147-BOS-LAX    BOS  LAX 2016-09-01 15:15:00 2016-09-01 21:45:00    2611.0
        3    197-BOS-ORD    BOS  ORD 2016-09-01 15:30:00 2016-09-01 18:24:00     867.0
        4    198-BOS-JFK    BOS  JFK 2016-09-01 13:10:00 2016-09-01 14:31:00     187.0
```

- The first column is the **index** of the DataFrame. The index provides a label for each row of the DataFrame.
- Right now, the index is sort of uninformative.
- Since each row corresponds to a flight, it would be nice if the index corresponded to the flight names.
- We can do this using the `.set_index()` method:

```
In [5]: # Set the index to the flight names
        df = df.set_index("FLIGHT")

        # Print the first 5 rows of df
        df.head()
```

```
Out[5]:             ORIGIN DEST            DEP_TIME            ARR_TIME  DISTANCE
        FLIGHT
        1-BOS-JFK      BOS  JFK 2016-09-01 06:00:00 2016-09-01 07:15:00     187.0
        40-BOS-ORD     BOS  ORD 2016-09-01 19:12:00 2016-09-01 22:02:00     867.0
        147-BOS-LAX    BOS  LAX 2016-09-01 15:15:00 2016-09-01 21:45:00    2611.0
        197-BOS-ORD    BOS  ORD 2016-09-01 15:30:00 2016-09-01 18:24:00     867.0
        198-BOS-JFK    BOS  JFK 2016-09-01 13:10:00 2016-09-01 14:31:00     187.0
```

- A column by itself is called a **Series**.
- You can select the Series `DEST` of the DataFrame `df` like this:

```
df["DEST"]
```

- So, to print the Series `DEST`, we could write:

```
In [6]: # Print the DEST column
        print(df["DEST"])
```

```
FLIGHT
1-BOS-JFK        JFK
40-BOS-ORD       ORD
147-BOS-LAX      LAX
197-BOS-ORD      ORD
198-BOS-JFK      JFK
85-BOS-JFK       JFK
252-BOS-ORD      ORD
333-BOS-LAX      LAX
1086-BOS-MIA     MIA
1094-BOS-DFW     DFW
307-BOS-PHX      PHX
1155-BOS-ORD     ORD
1172-BOS-MIA     MIA
1211-BOS-DFW     DFW
1274-BOS-MIA     MIA
166-BOS-LAX      LAX
175-BOS-DFW      DFW
1435-BOS-ORD     ORD
1404-BOS-ORD     ORD
1503-BOS-ORD     ORD
1509-BOS-MIA     MIA
1240-BOS-ORD     ORD
223-BOS-LAX      LAX
1006-BOS-MIA     MIA
1039-BOS-JFK     JFK
2303-BOS-DFW     DFW
2251-BOS-DFW     DFW
2253-BOS-ORD     ORD
2378-BOS-PHX     PHX
2454-BOS-MIA     MIA
               ...
```

```
1919-PVD-PHL    PHL
1961-PVD-CLT    CLT
2035-PVD-CLT    CLT
840-PVD-CLT     CLT
1653-PVD-CLT    CLT
1770-BUF-CLT    CLT
1789-BUF-CLT    CLT
1987-BUF-CLT    CLT
858-BUF-CLT     CLT
1793-SYR-CLT    CLT
2065-SYR-CLT    CLT
1797-MDT-CLT    CLT
1807-CHS-CLT    CLT
2063-CHS-CLT    CLT
1814-ALB-CLT    CLT
1895-ALB-CLT    CLT
2006-ALB-CLT    CLT
1832-PWM-CLT    CLT
1859-PWM-CLT    CLT
1861-GSO-CLT    CLT
1867-ROC-CLT    CLT
1868-ROC-CLT    CLT
2084-BOI-PHX    PHX
514-BOI-PHX     PHX
592-BOI-PHX     PHX
621-BOI-DFW     DFW
2357-LBB-DFW    DFW
2571-ANC-DFW    DFW
483-GEG-PHX     PHX
490-GEG-PHX     PHX
Name: DEST, Length: 2607, dtype: object
```

- You might want to click on the left of the output above — this will collapse the output so it doesn't take over your browser window.

Making the data easier to use

- We have successfully imported our data into Python!
- We could set up our shortest path problem using the DataFrame directly, but this would be a bit cumbersome.
- Let's take some additional steps that will make setting up our shortest path problem a bit easier.

- First, let's get a list of the flights. This will be useful, since the nodes in our shortest path problem correspond to the flights.
- In the DataFrame `df` we defined above, the index consists of the flights.
- We can get a list of the index values of `df` with `list(df.index.values)`.

```
In [7]: # List of flights
        flights = list(df.index.values)
```

- Let's check our work and inspect `flights`:

```
In [8]: # Print flights
        # Leaving this out - output is very, very long
        # print("Flights: {0}".format(flights))
```

- While we're here, let's make sure we have the right number of flights in the variable `flights`:

```
In [9]: print("Number of flights: {0}".format(len(flights)))
```

```
Number of flights: 2607
```

- We also want to easily access the origin, destination, departure time, arrival time, and distance for each flight.
- These correspond to the columns in our DataFrame `df`.
- We can convert DataFrame columns to dictionaries as using the `.to_dict()` on the column of interest, like this:

```
In [10]: # Convert columns to dictionaries
         origin = df['ORIGIN'].to_dict()
         destination = df['DEST'].to_dict()
         departure_time = df['DEP_TIME'].to_dict()
         arrival_time = df['ARR_TIME'].to_dict()
         distance = df['DISTANCE'].to_dict()
```

- As a result, we can access information about each flight through these dictionaries as follows:

```
In [11]: # Information about flight 1240-BOS-ORD
         print("Origin: {0}".format(origin['1240-BOS-ORD']))
         print("Destination: {0}".format(destination['1240-BOS-ORD']))
         print("Departure time: {0}".format(departure_time['1240-BOS-ORD']))
         print("Arrival time: {0}".format(arrival_time['1240-BOS-ORD']))
         print("Distance: {0}".format(distance['1240-BOS-ORD']))
```

```
Origin: BOS
Destination: ORD
Departure time: 2016-09-01 12:15:00
Arrival time: 2016-09-01 15:05:00
Distance: 867.0
```

Setting up the shortest path problem in networkx

- Now we're ready to set up the shortest path problem we formulated above.
- First, let's import `networkx` and `bellmanford` so we can use them:

```
In [12]: import networkx as nx
         import bellmanford as bf
```

Adding nodes

- Let's build the shortest path graph, starting with an empty directed graph:

```
In [13]: # Create empty NetworkX digraph
         G = nx.DiGraph()
```

- Next, let's create a "start" and "end" node.

```
In [14]: # Create start and end nodes
         G.add_node("start")
         G.add_node("end")
```

- Now, we need to add a node for each flight, or each row of our database.

```
In [15]: # Add a node for each flight
         for flight in flights:
             G.add_node(flight)
```

- The `.number_of_nodes()` method applied to a `networkx` graph — well, you can guess what it does. Or, you can just try it out:

```
In [16]: # Print number of nodes in G
         print(G.number_of_nodes())
```

```
2609
```

Adding edges

- Now we can go over every pair of flight nodes, and check if we need to add an edge between them.
  - Remember the length of these edges is the negative of the distance of the first flight.
- To add or subtract times, we need to use `pd.to_timedelta()` — here is the documentation.
  - For example, to subtract 30 minutes, we would write

    `some_time_variable - pd.to_timedelta(30, unit="m")`
- This might seem awkward, but if you think about it, working with dates and time *is* awkward — you need to keep track of different (non-base-10) units.

```
In [17]: # Iterate through every pair of flight nodes
         for first in flights:
             for second in flights:

                 # If the first flight arrives where the second flight departs...
                 if (destination[first] == origin[second]):

                     # And if the first flight arrives 45 minutes before the second flight
         leaves,
                     # add an edge from the first flight to the second
                     if (arrival_time[first] + pd.to_timedelta(45, unit="m") <
         departure_time[second]):
                         G.add_edge(first, second, length=-distance[first])
```

- Finally, we need to add edges:
    - from the start node to all flights departing from BWI, and
    - from all flights arriving at LAX to the end node.

```
In [18]: # Iterate through all flights
         for flight in flights:

             # If the flight departs from BWI,
             # add an edge from start to this flight
             if origin[flight] == "BWI":
                 G.add_edge("start", flight, length=0)

             # If the flight arrives at LAX,
             # add an edge from this flight to end
             if destination[flight] == "LAX":
                 G.add_edge(flight, "end", length=-distance[flight])
```

- Similar to `G.number_of_nodes()`, we can perform a sanity check with our work with `G.number_of_edges()`.

```
In [19]: # Print the number of edges in G
         print(G.number_of_edges())
```

```
158335
```

Solving the shortest path problem, interpreting the output

- Now that we have our directed graph set up, we can solve for the shortest path from the start node to the end node just like we did in the last lesson:

```
In [20]: # Solve the shortest path problem using Bellman-Ford
         length, nodes, negative_cycle = bf.bellman_ford(G, source="start", target="end",
         weight="length")

         # Print output from Bellman-Ford
         print("Negative cycle? {0}".format(negative_cycle))
         print("Shortest path length: {0}".format(length))
         print("Shortest path: {0}".format(nodes))
```

```
Negative cycle? False
Shortest path length: -8005.0
Shortest path: ['start', '1817-BWI-CLT', '658-CLT-LAS', '1584-LAS-PHX', '694-PHX-HNL',
'298-HNL-LAX', 'end']
```

- What does the output tell us about how to solve Professor Wright's problem?

- The length of the shortest path is the negative of the maximum total distance Professor Wright can travel from BWI to LAX. In this case, the maximum total distance is 8005 miles.

- The nodes in the shortest path tells us which flights Professor Wright should take:
  - 1817 from BWI to CLT
  - 658 from CLT to LAS
  - 1584 from LAS to PHX
  - 694 from PHX to HNL
  - 298 from HNL to LAX

On your own...

Suppose Professor Wright wants to find the longest itinerary from IAD (Washington DC - Dulles) to SAN (San Diego) instead.

In the cell below, write the code that sets up and solves the shortest path formulation for her problem from start to finish.

In the cell after, describe in words what the output from the Bellman-Ford algorithm tells you about how to solve Professor Wright's problem.

```
In [21]: # Import packages
         import pandas as pd
         import networkx as nx
         import bellmanford as bf

         # Read csv file into a DataFrame
         # Designate departure and arrival time columns as dates
         df = pd.read_csv('aa_domestic_flights.csv', parse_dates=['DEP_TIME', 'ARR_TIME'])

         # Create empty networkx digraph
         G = nx.DiGraph()

         # Create start and end nodes
         G.add_node("start")
         G.add_node("end")

         # Add a node for each flight
         for flight in flights:
             G.add_node(flight)

         # Iterate through every pair of flight nodes
         for first in flights:
             for second in flights:

                 # If the first flight arrives where the second flight departs...
                 if (destination[first] == origin[second]):

                     # And if the first flight arrives 45 minutes before the second flight
         leaves,
                     # add an edge from the first flight to the second
                     if (arrival_time[first] + pd.to_timedelta(45, unit="m") <
         departure_time[second]):
                         G.add_edge(first, second, length=-distance[first])

         # Iterate through all flights
         for flight in flights:

             # If the flight departs from IAD,
             # add an edge from start to this flight
             if origin[flight] == "IAD":
                 G.add_edge("start", flight, length=0)

             # If the flight arrives at SAN,
             # add an edge from this flight to end
             if destination[flight] == "SAN":
                 G.add_edge(flight, "end", length=-distance[flight])

         # Solve the shortest path problem using Bellman-Ford
         length, nodes, negative_cycle = bf.bellman_ford(G, source="start", target="end",
         weight="length")

         # Print output from Bellman-Ford
         print("Negative cycle? {0}".format(negative_cycle))
         print("Shortest path length: {0}".format(length))
         print("Shortest path: {0}".format(nodes))
```

```
Negative cycle? False
Shortest path length: -6005.0
Shortest path: ['start', '2636-IAD-LAX', '2503-LAX-ORD', '2375-ORD-DFW', '435-DFW-SAN',
'end']
```

- The length of the shortest path is the negative of the maximum total distance Professor Wright can travel from BWI to LAX. In this case, the maximum total distance is 6005 miles.
- The nodes in the shortest path tells us which flights Professor Wright should take:
  - 2636 from IAD to LAX
  - 2503 from LAX to ORD
  - 2375 from ORD to DFW
  - 435 from DFW to SAN