Lesson 6.

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 are a few resources:
  - Here is the official documentation for pandas.
  - Chris Albon's notes are also a good resource on how to get things done with pandas (look in the *Data Wrangling* section).
- In this lesson, we'll use pandas in a very basic way to help us 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
```

- Another useful method is `.describe()`.
- By default, `.describe()` only provides summary statistics for the columns with numeric data.
- To get summary statistics for all the columns, include the argument `include="all"`, like this:

```
In [5]: # Get summary statistics for all columns in df
        df.describe(include="all")
```

```
Out[5]:            FLIGHT ORIGIN  DEST             DEP_TIME             ARR_TIME  \
        count         2607   2607  2607                 2607                 2607
        unique        2607     94    94                  643                 1046
        top     32-LAX-JFK    DFW   DFW  2016-09-01 07:00:00  2016-09-01 16:45:00
        freq             1    408   407                   43                   12
        first          NaN    NaN   NaN  2016-09-01 02:15:00  2016-09-01 06:01:00
        last           NaN    NaN   NaN  2016-09-02 03:50:00  2016-09-02 09:33:00
        mean           NaN    NaN   NaN                  NaN                  NaN
        std            NaN    NaN   NaN                  NaN                  NaN
        min            NaN    NaN   NaN                  NaN                  NaN
        25%            NaN    NaN   NaN                  NaN                  NaN
        50%            NaN    NaN   NaN                  NaN                  NaN
        75%            NaN    NaN   NaN                  NaN                  NaN
        max            NaN    NaN   NaN                  NaN                  NaN

                   DISTANCE
        count   2607.000000
        unique          NaN
        top             NaN
        freq            NaN
        first           NaN
        last            NaN
        mean     986.709628
        std      645.910443
        min       83.000000
        25%      507.000000
        50%      868.000000
        75%     1272.000000
        max     3784.000000
```

- 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"])
```

```
0       JFK
1       ORD
2       LAX
3       ORD
4       JFK
5       JFK
6       ORD
7       LAX
8       MIA
9       DFW
10      PHX
11      ORD
```

```
12      MIA
13      DFW
14      MIA
15      LAX
16      DFW
17      ORD
18      ORD
19      ORD
20      MIA
21      ORD
22      LAX
23      MIA
24      JFK
25      DFW
26      DFW
27      ORD
28      PHX
29      MIA
       ...
2577    PHL
2578    CLT
2579    CLT
2580    CLT
2581    CLT
2582    CLT
2583    CLT
2584    CLT
2585    CLT
2586    CLT
2587    CLT
2588    CLT
2589    CLT
2590    CLT
2591    CLT
2592    CLT
2593    CLT
2594    CLT
2595    CLT
2596    CLT
2597    CLT
2598    CLT
2599    PHX
2600    PHX
2601    PHX
2602    DFW
2603    DFW
2604    DFW
2605    PHX
2606    PHX
Name: DEST, dtype: object
```

Setting up the shortest path problem in networkx

- Now that we can access the flight database in Python, we can use its contents to setup the shortest path problem we formulated above.
- First, let's import `networkx` and `bellmanford` so we can use them:

```
In [7]: import networkx as nx
        import bellmanford as bf
```

Creating a list of flights

- It will be useful to create a variable `flights` containing a list of all the flights.
- What part of the dataset contains this information?

We need to look at the `FLIGHT` column of the dataset.

- From the `.describe()` output above, we see that the flights in `df["FLIGHT"]` are unique.
- We can convert the Series `df["FLIGHT"]` to a list with the function `list()`.
    - Then we can use the list methods we learned about earlier, such as `.append()`, if necessary.

```
In [8]: # Take the FLIGHT column from df, convert it to a list
        flights = list(df["FLIGHT"])
```

- It's a good idea to make sure nothing funny happened — let's inspect the variable `flights` we just created:

```
In [9]: # Print flights
        print("Flights: {0}".format(flights))
```

```
Flights: ['1-BOS-JFK', '40-BOS-ORD', '147-BOS-LAX', '197-BOS-ORD', '198-BOS-JFK', '85-BOS-JFK', '252-BOS-ORD', '333-BOS-LA
```

- 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.
- Let's also make sure we have the right number of flights in the variable `flights`:

```
In [10]: print("Number of flights: {0}".format(len(flights)))
```

```
Number of flights: 2607
```

Creating a list of airports

- It will also be useful to have a variable `airports` containing a list of all the airports.
- What part of the dataset contains this information?

We need to look at the `ORIGIN` and `DEST` columns of the dataset.

- We can create the list `airports` like this:

```
In [11]: # Convert the ORIGIN and DEST columns from df into sets,
         # take their union, convert to a list
         airports = list(set(df["ORIGIN"]) | set(df["DEST"]))
```

- Um… what does this do??
- Let's try a smaller example and look at what's going on step-by-step.
- Pretend that `A` and `B` defined below are the `ORIG` and `DEST` columns from `df`

```
In [12]: # Pretend that A and B are the ORIG and DEST columns from df
         A = ['BWI', 'BWI', 'ORD', 'ORD']
         B = ['LAX', 'ORD', 'SFO', 'LAX']
```

- In Python, a **set** is an unordered collection of unique elements, just like the usual mathematical definition.
- `set(A)` takes all entries `A` and converts it into a set. This eliminates all duplicates within `A`.
- Same goes for `set(B)`.

```
In [13]: # Convert A and B into sets, print them out
         print(set(A))
         print(set(B))
```

```
{'BWI', 'ORD'}
{'SFO', 'ORD', 'LAX'}
```

- The `|` operator takes the **union** of the sets, like this:

```
In [14]: # Take the union of set(A) and set(B), print it out
         print(set(A) | set(B))
```

```
{'BWI', 'SFO', 'ORD', 'LAX'}
```

- This is almost what we want: we have a list of all the airports, but…
- Sets are similar to lists, but have their own methods. We can turn the set into a list with the function `list()`, like this:

```
In [15]: # Print out the union of set(A) and set(B), converted to a list
         print(list(set(A) | set(B)))
```

```
['BWI', 'SFO', 'ORD', 'LAX']
```

- See how that works? That's why `airports` defined above contains a list of all the airports in our dataset.
- Let's make sure everything looks OK with `airports`:

```
In [16]: # Print list of airports
         print('Airports: {0}'.format(airports))

         # Print number of airports
         print('Number of airports: {0}'.format(len(airports)))
```

```
Airports: ['MFE', 'SLC', 'SNA', 'STX', 'DEN', 'FAT', 'MIA', 'SFO', 'HNL', 'MCO', 'MDT', 'FLL', 'STL', 'IAD', 'TPA', 'MEM',
Number of airports: 94
```

Adding nodes with attributes

- Now we're ready to build the shortest path graph. Let's start with an empty directed graph:

```
In [17]: # Create empty NetworkX digraph
         G = nx.DiGraph()
```

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

```
In [18]: # 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.
- We can quickly iterate through the rows of a DataFrame using the `.itertuples()` method:

```
for row in df.itertuples():
    # Put some code here
    # row.COLUMN_NAME = value of column COLUMN_NAME in the current row
```

- So we can add a node for each flight like this:

```
In [19]: # Add a node for each flight
         for row in df.itertuples():
             G.add_node(row.FLIGHT, origin=row.ORIGIN, dest=row.DEST, dep_time=row.DEP_TIME, arr_time=row.ARR_TIME,
                        distance=row.DISTANCE)
```

- Wait —

  ```
  G.add_node(row.FLIGHT)
  ```

  adds a node whose name is the value of `row.FLIGHT`. What is all the other stuff?

- Remember in the last lesson when we added the "length" attribute to each edge? Like this?

```
G.add_edge(1, 2, length=9)
```

- We can add attributes to nodes as well.
- The code above adds attributes called `origin`, `dest`, `dep_time`, `arr_time`, and `distance` to each node.
  - This will be handy later.
- To access a particular attribute of a node, we write something like this:

```
In [20]: # print the departure time of flight "1-BOS-JFK"
         print(G.node["1-BOS-JFK"]["dep_time"])
```

```
2016-09-01 06:00:00
```

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

```
2609
```

Adding edges

- Now we can check 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 [22]: # 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 (G.node[first]["dest"] == G.node[second]["origin"]):

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

- 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 [23]: # Iterate through all flights
         for flight in flights:

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

             # If the flight arrives at LAX,
```

```
            # add an edge from this flight to end
            if G.node[flight]["dest"] == "LAX":
                G.add_edge(flight, "end", length=-G.node[flight]["distance"])
```

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

```
In [24]: # 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 [25]: # 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 shortest path length gives us the *negative* of the maximum possible total distance Professor Wright can travel on a feasible sequence of flights from BWI to LAX, which in this case, is 8005 miles.

- The nodes in the shortest path give the sequence of flights from BWI to LAX with the maximum possible total distance, which in this case is:

    1. Flight 1817, BWI-CLT
    2. Flight 658, CLT-LAS
    3. Flight 1584, LAS-PHX
    4. Flight 694, PHX-HNL
    5. Flight 298, HNL-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 [26]:
```
# 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 row in df.itertuples():
    G.add_node(row.FLIGHT, origin=row.ORIGIN, dest=row.DEST, dep_time=row.DEP_TIME, arr_time=row.ARR_TIME,
               distance=row.DISTANCE)

# 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 (G.node[first]["dest"] == G.node[second]["origin"]):

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

# Iterate through all flights
for flight in flights:

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

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

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

```python
        # 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 shortest path length gives us the *negative* of the maximum possible total distance Professor Wright can travel on a feasible sequence of flights from IAD to SAN, which in this case, is 6005 miles.
- The nodes in the shortest path give the sequence of flights from IAD to SAN with the maximum possible total distance, which in this case is:
  1. Flight 2636, IAD-LAX
  2. Flight 2503, LAX-ORD
  3. Flight 2375, ORD-DFW
  4. Flight 435, DFW-SAN