Markov chain analysis of the US airport network

Laura M. Schultz

One way to view the airline transportation infrastructure is in the form of a directed *network* or *graph*, in which vertices are airports and edges are the direct-flight segments that connect them. For instance, if there is a direct flight from Atlanta's Hartsfield-Jackson International Airport ("ATL") to Los Angeles International Airport ("LAX"), then the airport network would have a directed edge from ATL to LAX.

Given the airport network, one question we might ask is, which airports are most critical to disruption of the overall network? That is, if an airport is shut down, thereby leading to all inbound and outbound flights being cancelled, will that catastrophic event have a big impact or a small impact on the overall network?

We would expect "importance" to be related to whether an airport has lots of inbound or outgoing connections. In graph lingo, that's also called the *degree* of a vertex or node. But if there are multiple routes that can work around a highly connected hub (i.e., a vertex with a high indegree or outdegree), that might not be the case. So my goal is to use a PageRank-like scheme to produce a ranking and compare it to ranking based on degree.

As it happens, the US Bureau of Transportation Statistics collects data on all flights originating or arriving in the United States. In this notebook, I'll use this dataset to build an airport network and then use Markov chain analysis to rank the networks by some measure of "criticality."

> Sources: This notebook is adapted from the following: https://www.mongodb.com/blog/post/pagerank-on-flights-dataset (https://www.mongodb.com/blog/post/pagerank-on-flights-dataset). The dataset I used was taken from the repository available here: https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236 (https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236)

The formal analysis problem

Let's model the analysis problem as follows.

Consider a "random flyer" to be a person who arrives at an airport i, and then randomly selects any direct flight that departs from i and arrives at j. We refer to the direct flight from i to j as the *flight segment* $i \rightarrow j$. Upon arriving at j, the flyer repeats the process of randomly selecting a new segment, $j \rightarrow k$. He or she repeats this process forever.

Let $x_i(t)$ be the probability that the flyer is at airport i at time t. Take t to be an integer count corresponding to the number of flight segments that the flyer has taken so far, starting at $t = 0$. Let p_{ij} be the probability of taking segment $i \rightarrow j$, where $p_{ij} = 0$ means the segment $i \rightarrow j$ is unavailable or does not exist. If there are n airports in all, numbered from 0 to $n - 1$, then the probability that the flyer will be at airport i at time $t + 1$, given all the probabilities at time t, is

$$x_i(t+1) = \sum_{j=0}^{n-1} p_{ji} \cdot x_j(t).$$

Let $P \equiv [p_{ij}]$ be the matrix of transition probabilities and $x(t) = [x_i(t)]$ the column vector of prior probabilities. Then we can write the above more succinctly for all airports as the matrix-vector product,

$$x(t+1) = P^T x(t).$$

Since P is a probability transition matrix then there exists a steady-state distribution, x^* , which is the limit of $x(t)$ as t goes to infinity:

$$\lim_{t \rightarrow \infty} x(t) = x^* \equiv [x_i^*].$$

The larger x_i^* , the more likely it is that the random flyer is to be at airport i in the steady state. Therefore, we can take the "importance" or "criticality" of airport i in the flight network to be its steady-state probability, x_i^* .

Thus, my data pre-processing goal is to construct P and our analysis goal is to compute the steady-state probability distribution, x^* , for a first-order Markov chain system.

In this notebook, I will use Pandas for preprocessing the raw data and SciPy's sparse matrix libraries to implement the analysis.

> One of the cells below defines a function, `spy()`, that can be used to visualize the non-zero structure of a sparse matrix.

In [1]:
```
import numpy as np
import scipy as sp
import scipy.sparse

import pandas as pd
```

In [2]:
```
import matplotlib.pyplot as plt
%matplotlib inline

def spy(A, figsize=(6, 6), markersize=0.5):
    """Visualizes a sparse matrix."""
    fig = plt.figure(figsize=figsize)
    plt.spy(A, markersize=markersize)
    plt.show()
```

In [3]:
```
from IPython.display import display, Markdown # For pretty-printing ti
bbles
```

In [4]:
```
def canonicalize_tibble(X):
    var_names = sorted(X.columns)
    Y = X[var_names].copy()
    Y.sort_values(by=var_names, inplace=True)
    Y.reset_index(drop=True, inplace=True)
    return Y

def tibbles_are_equivalent (A, B):
    A_canonical = canonicalize_tibble(A)
    B_canonical = canonicalize_tibble(B)
    cmp = A_canonical.eq(B_canonical)
    return cmp.all().all()
```

Downloading, unpacking, and exploring the data

In [5]:
```
import requests
import os
import hashlib
import io

def on_vocareum():
```

```
    return os.path.exists('.voc')

def download(file, local_dir="", url_base=None, checksum=None):
    local_file = "{}{}".format(local_dir, file)
    if not os.path.exists(local_file):
        if url_base is None:
            url_base = "https://cse6040.gatech.edu/datasets/"
        url = "{}{}".format(url_base, file)
        print("Downloading: {} ...".format(url))
        r = requests.get(url)
        with open(local_file, 'wb') as f:
            f.write(r.content)

    if checksum is not None:
        with io.open(local_file, 'rb') as f:
            body = f.read()
            body_checksum = hashlib.md5(body).hexdigest()
            assert body_checksum == checksum, \
                "Downloaded file '{}' has incorrect checksum: '{}' ins
tead of '{}'".format(local_file,
                body_checksum, checksum)
    print("'{}' is ready!".format(file))

if on_vocareum():
    URL_BASE = "https://cse6040.gatech.edu/datasets/us-flights/"
    DATA_PATH = "../resource/asnlib/publicdata/"
else:
    URL_BASE = "https://github.com/cse6040/labs-fa17/raw/master/lab11-
markov_chains/"
    DATA_PATH = ""

datasets = {'L_AIRPORT_ID.csv': 'e9f250e3c93d625cce92d08648c4bbf0',
            'L_CITY_MARKET_ID.csv': 'f430a16a5fe4b9a849accb5d332b2bb8'
,
            'L_UNIQUE_CARRIERS.csv': 'bebe919e85e2cf72e7041dbf1ae5794e
',
            'us-flights--2017-08.csv': 'eeb259c0cdd00ff1027261ca0a7c03
32',
            'flights_atl_to_lax_soln.csv': '4591f6501411de90af72693cdb
cc08bb',
            'origins_top10_soln.csv': 'de85c321c45c7bf65612754be456708
6',
            'dests_soln.csv': '370f4c632623616b3bf26b6f79993fe4',
            'dests_top10_soln.csv': '4c7dd7edf48c4d62466964d6b8c14184'
,
            'segments_soln.csv': '516a78d2d9d768d78bfb012b77671f38',
            'segments_outdegree_soln.csv': 'b29d60151c617ebafd3a1c5854
1477c8'
           }

for filename, checksum in datasets.items():
    download(filename, local_dir=DATA_PATH, url_base=URL_BASE, checksu
m=checksum)
```

```
print("\n(All data appears to be ready.)")
```

```
'L_AIRPORT_ID.csv' is ready!
'L_CITY_MARKET_ID.csv' is ready!
'L_UNIQUE_CARRIERS.csv' is ready!
'us-flights--2017-08.csv' is ready!
'flights_atl_to_lax_soln.csv' is ready!
'origins_top10_soln.csv' is ready!
'dests_soln.csv' is ready!
'dests_top10_soln.csv' is ready!
'segments_soln.csv' is ready!
'segments_outdegree_soln.csv' is ready!

(All data appears to be ready.)
```

Airport codes. Let's start with the airport codes.

In [6]:
```
airport_codes = pd.read_csv("{}{}".format(DATA_PATH, 'L_AIRPORT_ID.csv
'))
airport_codes.head()
```

Out[6]:

	Code	Description
0	10001	Afognak Lake, AK: Afognak Lake Airport
1	10003	Granite Mountain, AK: Bear Creek Mining Strip
2	10004	Lik, AK: Lik Mining Camp
3	10005	Little Squaw, AK: Little Squaw Airport
4	10006	Kizhuyak, AK: Kizhuyak Bay

Flight segments. Next, I loaded a file that contains all of US flights that were scheduled for August 2017.

In [7]:
```
flights = pd.read_csv('{}{}'.format(DATA_PATH, 'us-flights--2017-08.cs
v'))
print("Number of flight segments: {} [{:.1f} million]".format (len(fli
ghts), len(flights)*1e-6))
del flights['Unnamed: 7'] # Cleanup extraneous column
flights.head()
```

```
Number of flight segments: 510451 [0.5 million]
```

Out[7]:

	FL_DATE	UNIQUE_CARRIER	FL_NUM	ORIGIN_AIRPORT_ID	ORIGIN_CITY_MARKE
0	2017-08-01	DL	2	12478	31703
1	2017-08-01	DL	4	12889	32211
2	2017-08-01	DL	6	12892	32575
3	2017-08-01	DL	7	14869	34614
4	2017-08-01	DL	10	11292	30325

Each row of this tibble is a *(direct) flight segment*, that is, a flight that left some origin and arrived at a destination on a certain date. As noted earlier, these segments cover a one-month period (August 2017).

The first step is to familiarize myself with the data.

I began by using the `airport_codes` data frame to figure out the integer airport codes (not the three-letter codes) for Atlanta's Hartsfield-Jackson International (ATL) and Los Angeles International (LAX). I stored these codes in variables named `ATL_ID` and `LAX_ID`, respectively.

Next, I determined all of the direct flight segments that originated at ATL and traveled to LAX. I stored the result in a dataframe named `flights_atl_to_lax`, which is the corresponding subset of rows from `flights`.

In [8]:

```
# PART A) Define `ATL_ID` and `LAX_ID` to correspond to the
# codes in `airport_codes` for ATL and LAX, respectively.

ATL_row = airport_codes[airport_codes['Description'].str.contains("Har
tsfield-Jackson")]
print(ATL_row)
ATL_ID = 10397


LAX_row = airport_codes[airport_codes['Description'].str.contains("Los
Angeles International")]
print(LAX_row)
LAX_ID = 12892


# Print the descriptions of the airports with their IDs:
ATL_DESC = airport_codes[airport_codes['Code'] == ATL_ID]['Description
'].iloc
LAX_DESC = airport_codes[airport_codes['Code'] == LAX_ID]['Description
'].iloc
print("{}: ATL -- {}".format(ATL_ID, ATL_DESC))
print("{}: LAX -- {}".format(LAX_ID, LAX_DESC))
```

```
      Code                                       Description
373  10397  Atlanta, GA: Hartsfield-Jackson Atlanta Intern...
       Code                                 Description
2765  12892  Los Angeles, CA: Los Angeles International
10397: ATL -- <pandas.core.indexing._iLocIndexer object at 0x1110606
88>
12892: LAX -- <pandas.core.indexing._iLocIndexer object at 0x111060e
58>
```

In [9]:
```
# PART B) Construct `flights_atl_to_lax`
#

flights_atl_to_lax = flights[(flights['ORIGIN_AIRPORT_ID'] == 10397) &
                             (flights['DEST_AIRPORT_ID'] == 12892)]

# Displays a few of my results
print("My code found {} flight segments.".format(len(flights_atl_to_la
x)))
display(flights_atl_to_lax.head())
```

```
My code found 586 flight segments.
```

	FL_DATE	UNIQUE_CARRIER	FL_NUM	ORIGIN_AIRPORT_ID	ORIGIN_CITY_MAR
64	2017-08-01	DL	110	10397	30397
165	2017-08-01	DL	370	10397	30397
797	2017-08-01	DL	1125	10397	30397
806	2017-08-01	DL	1133	10397	30397
858	2017-08-01	DL	1172	10397	30397

Aggregation. Observe that an (origin, destination) pair may appear many times. That's because the dataset includes a row for *every* direct flight that occurred historically and there may have been multiple such flights on a given day.

However, for the purpose of this analysis, I will simplify the problem by collapsing *all* historical segments $i \to j$ into a single segment. Moreover, I will do so in a way that preserves the number of times the segment occurred (i.e., the number of rows containing the segment).

To accomplish this task, the following code cell uses the [`groupby()` (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.groupby.html)](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.groupby.html) function available for Pandas tables and the [`count()` (http://pandas.pydata.org/pandas-docs/stable/groupby.html)](http://pandas.pydata.org/pandas-docs/stable/groupby.html) aggregator in three steps:

1. It considers just the flight date, origin, and destination columns.
2. It *logically* groups the rows having the same origin and destination, using `groupby()`.
3. It then aggregates the rows, counting the number of rows in each (origin, destination) group.

In [10]:
```
flights_cols_subset = flights[['FL_DATE', 'ORIGIN_AIRPORT_ID', 'DEST_A
IRPORT_ID']]
segment_groups = flights_cols_subset.groupby(['ORIGIN_AIRPORT_ID', 'DE
ST_AIRPORT_ID'], as_index=False)
segments = segment_groups.count()
segments.rename(columns={'FL_DATE': 'FL_COUNT'}, inplace=True)
segments.head()
```

Out[10]:

	ORIGIN_AIRPORT_ID	DEST_AIRPORT_ID	FL_COUNT
0	10135	10397	77
1	10135	11433	85
2	10135	13930	18
3	10140	10397	93
4	10140	10423	4

Finally, I verified that the counts are all at least 1.

In [11]:
```
assert (segments['FL_COUNT'] > 0).all()
```

Actual (as opposed to "all possible") origins and destinations. Although there are many possible airport codes stored in the `airport_codes` dataframe (over six thousand), only a subset appear as actual origins in the data. The following code cell determines the actual origins and prints their number.

In [12]:
```
origins = segments[['ORIGIN_AIRPORT_ID', 'FL_COUNT']].groupby('ORIGIN_
AIRPORT_ID', as_index=False).sum()
origins.rename(columns={'FL_COUNT': 'ORIGIN_COUNT'}, inplace=True)
print("Number of actual origins:", len(origins))
origins.head()
```

```
Number of actual origins: 300
```

Out[12]:

	ORIGIN_AIRPORT_ID	ORIGIN_COUNT
0	10135	180
1	10140	1761
2	10141	62
3	10146	41
4	10154	176

To get an idea of what airports are likely to be the most important in my Markov chain analysis, I will rank the airports by the total number of *outgoing* segments, i.e., flight segments that originate at the airport.

Here, I have constructed a dataframe, `origins_top10`, containing the top 10 airports in descending order of outgoing segments. This dataframe has three columns:

- `ID`: The ID of the airport
- `Count`: Number of outgoing segments.
- `Description`: The plaintext descriptor for the airport that comes from the `airport_codes` dataframe.

In [13]:
```
import numpy as np

origins_A = origins.rename(columns = {'ORIGIN_AIRPORT_ID' : 'Code', 'O
RIGIN_COUNT' : 'Count'})
origins_A.head()

origins_B = origins_A.merge(airport_codes, on = ['Code'])
origins_C = origins_B.rename(columns = {'Code' : 'ID'})

origins_D = origins_C.sort_values(by=('Count'), ascending=False)

origins_top10 = origins_D.head(10)

origins_top10 = origins_top10.reset_index(drop=True)

# Prints the top 10, according to my calculation:
origins_top10
```

Out[13]:

	ID	Count	Description
0	10397	31899	Atlanta, GA: Hartsfield-Jackson Atlanta Intern...
1	13930	25757	Chicago, IL: Chicago O'Hare International
2	11292	20891	Denver, CO: Denver International
3	12892	19399	Los Angeles, CA: Los Angeles International
4	14771	16641	San Francisco, CA: San Francisco International
5	11298	15977	Dallas/Fort Worth, TX: Dallas/Fort Worth Inter...
6	14747	13578	Seattle, WA: Seattle/Tacoma International
7	12889	13367	Las Vegas, NV: McCarran International
8	14107	13040	Phoenix, AZ: Phoenix Sky Harbor International
9	13487	12808	Minneapolis, MN: Minneapolis-St Paul Internati...

The preceding code computed a tibble, `origins`, containing all the unique origins and their number of outgoing flights. Here, I have written code to compute a new tibble, `dests`, which contains all unique destinations and their number of *incoming* flights. The columns are named `DEST_AIRPORT_ID` (airport code) and `DEST_COUNT` (number of direct inbound segments). I have printed the first five rows of my tibble below.

In [14]:
```
dests = segments[['DEST_AIRPORT_ID', 'FL_COUNT']].groupby('DEST_AIRPOR
T_ID', as_index = False).sum()
dests.rename(columns = {'FL_COUNT' : 'DEST_COUNT'}, inplace = True)

print("Number of unique destinations:", len(dests))
dests.head()
```

```
Number of unique destinations: 300
```

Out[14]:

	DEST_AIRPORT_ID	DEST_COUNT
0	10135	179
1	10140	1763
2	10141	62
3	10146	40
4	10154	176

Next, I computed a tibble, `dests_top10`, containing the top 10 destinations (i.e., rows of `dests`) by inbound flight count. The column names are the same as `origins_top10` and the rows are sorted in decreasing order by count.

In [15]:
```
dests_A = dests.rename(columns = {'DEST_AIRPORT_ID' : 'Code', 'DEST_CO
UNT' : 'Count'})
dests_A.head()

dests_B = dests_A.merge(airport_codes, on = ['Code'])
dests_C = dests_B.rename(columns = {'Code' : 'ID'})

dests_D = dests_C.sort_values(by=('Count'), ascending=False)

dests_top10 = dests_D.head(10)

dests_top10 = dests_top10.reset_index(drop=True)


print("My computed top 10 destinations:")
dests_top10
```

```
My computed top 10 destinations:
```

Out[15]:

	ID	Count	Description
0	10397	31901	Atlanta, GA: Hartsfield-Jackson Atlanta Intern...
1	13930	25778	Chicago, IL: Chicago O'Hare International
2	11292	20897	Denver, CO: Denver International
3	12892	19387	Los Angeles, CA: Los Angeles International
4	14771	16651	San Francisco, CA: San Francisco International
5	11298	15978	Dallas/Fort Worth, TX: Dallas/Fort Worth Inter...
6	14747	13582	Seattle, WA: Seattle/Tacoma International
7	12889	13374	Las Vegas, NV: McCarran International
8	14107	13039	Phoenix, AZ: Phoenix Sky Harbor International
9	13487	12800	Minneapolis, MN: Minneapolis-St Paul Internati...

Having confirmed that the number of actual origins equals the number of actual destinations, I will store this value to use later.

In [16]:
```
n_actual = len(set(origins['ORIGIN_AIRPORT_ID']) | set(dests['DEST_AIR
PORT_ID']))
print("Number of actual locations (whether origin or destination):", n
_actual)
```

```
Number of actual locations (whether origin or destination): 300
```

Constructing the state-transition matrix

Now that I have cleaned up the data, I need to prepare it for subsequent analysis. I will start by constructing the *probability state-transition matrix* for the airport network. I will name this matrix by $P \equiv [p_{ij}]$, where p_{ij} is the conditional probability that a random flyer departs from airport i and arrives at airport j given that he or she is currently at airport i.

To build P, I will use SciPy's sparse matrix facilities. To do so, I will need to carry out the following two steps:

1. *Map airport codes to matrix indices.* An `m`-by-`n` sparse matrix in SciPy uses the zero-based values 0, 1, ..., `m`-1 and 0, ..., `n`-1 to refer to row and column indices. Therefore, I will need to map the airport codes to such index values.
2. *Derive weights, $p{ij}.* $_{I}\ willneedtodecidehowtodetermine$ p_{ij}$.

Step 1: Mapping airport codes to integers. Luckily, I already have a code-to-integer mapping, which is in the column `airport_codes['Code']` mapped to the dataframe's `index`.

As a first step, I will make note of the number of airports, which is just the largest index value.

In [17]:
```
n_airports = airport_codes.index.max() + 1
print("Note: There are", n_airports, "airports.")
```

```
Note: There are 6436 airports.
```

Next, I added another column to `segments` called `ORIGIN_INDEX`, which will hold the id corresponding to the origin:

In [18]:
```
# Recall:
segments.columns
```

Out[18]:
```
Index(['ORIGIN_AIRPORT_ID', 'DEST_AIRPORT_ID', 'FL_COUNT'], dtype='o
bject')
```

In [19]:
```
# Extract the `Code` column and index from `airport_codes`, storing th
em in
# a temporary tibble with new names, `ORIGIN_AIRPORT_ID` and `ORIGIN_I
NDEX`.
origin_indices = airport_codes[['Code']].rename(columns={'Code': 'ORIG
IN_AIRPORT_ID'})
origin_indices['ORIGIN_INDEX'] = airport_codes.index

# Since I might run this code cell multiple times, the following
# check prevents `ORIGIN_ID` from appearing more than once.
if 'ORIGIN_INDEX' in segments.columns:
    del segments['ORIGIN_INDEX']

# Perform the merge as a left-join of `segments` and `origin_ids`.
segments = segments.merge(origin_indices, on='ORIGIN_AIRPORT_ID', how=
'left')
segments.head()
```

Out[19]:

	ORIGIN_AIRPORT_ID	DEST_AIRPORT_ID	FL_COUNT	ORIGIN_INDEX
0	10135	10397	77	119
1	10135	11433	85	119
2	10135	13930	18	119
3	10140	10397	93	124
4	10140	10423	4	124

Analogous to the preceding procedure, I also created a new column called `segments['DEST_INDEX']` to hold the integer index of each segment's *destination*.

In [20]:
```
dest_indices = airport_codes[['Code']].rename(columns={'Code': 'DEST_A
IRPORT_ID'})
dest_indices['DEST_INDEX'] = airport_codes.index

if 'DEST_INDEX' in segments.columns:
    del segments['DEST_INDEX']

segments = segments.merge(dest_indices, on='DEST_AIRPORT_ID', how='lef
t')

# Visually inspect my result:
segments.head()
```

Out[20]:

	ORIGIN_AIRPORT_ID	DEST_AIRPORT_ID	FL_COUNT	ORIGIN_INDEX	DEST_INDEX
0	10135	10397	77	119	373
1	10135	11433	85	119	1375
2	10135	13930	18	119	3770
3	10140	10397	93	124	373
4	10140	10423	4	124	399

Step 2: Computing edge weights. Armed with the preceding mapping, I can now determine each segment's transition probability, or "weight," p_{ij}.

For each origin i, let d_i be the number of outgoing edges, or *outdegree*. Note that this value is *not* the same as the total number of (historical) outbound *segments*; rather, let's take d_i to be just the number of airports reachable directly from i. For instance, consider all flights departing the airport whose airport code is 10135:

In [21]:
```
display(airport_codes[airport_codes['Code'] == 10135])

abe_segments = segments[segments['ORIGIN_AIRPORT_ID'] == 10135]
display(abe_segments)

print("Total outgoing segments:", abe_segments['FL_COUNT'].sum())
```

	Code	Description
119	10135	Allentown/Bethlehem/Easton, PA: Lehigh Valley ...

	ORIGIN_AIRPORT_ID	DEST_AIRPORT_ID	FL_COUNT	ORIGIN_INDEX	DEST_INDEX
0	10135	10397	77	119	373
1	10135	11433	85	119	1375
2	10135	13930	18	119	3770

```
Total outgoing segments: 180
```

In [22]:
```
k_ABE = abe_segments['FL_COUNT'].sum()
d_ABE = len(abe_segments)
i_ABE = abe_segments['ORIGIN_AIRPORT_ID'].values[0]

display(Markdown('''
Though `ABE` has {} outgoing segments,
its outdegree or number of outgoing edges is just {}.
Thus, `ABE`, whose airport id is $i={}$, has $d_{{{}}} = {}$.
'''.format(k_ABE, d_ABE, i_ABE, i_ABE, d_ABE)))
```

Though `ABE` has 180 outgoing segments, its outdegree or number of outgoing edges is just 3. Thus, `ABE`, whose airport id is $i = 10135$, has $d_{10135} = 3$.

Next, I have added a new column named `OUTDEGREE` to the `segments` tibble that holds the outdegrees, $\{d_i\}$. That is, for each row whose airport *index* (as opposed to code) is i, its entry of `OUTDEGREE` should be d_i.

For instance, the rows of segments corresponding to airport ABE (code 10135 and matrix index 119) would look like this:

ORIGIN_AIRPORT_ID	DEST_AIRPORT_ID	FL_COUNT	ORIGIN_INDEX	DEST_INDEX	OUTDEGREE
10135	10397	77	119	373	3
10135	11433	85	119	1375	3
10135	13930	18	119	3770	3

In [23]:

```
# This `if` removes an existing `OUTDEGREE` column
# in case I run this cell more than once.
if 'OUTDEGREE' in segments.columns:
    del segments['OUTDEGREE']

segments['OUTDEGREE'] = segments.groupby('ORIGIN_INDEX')['ORIGIN_INDEX
'].transform('count')


# Visually inspect the first ten rows of my result:
segments.head(10)
```

Out[23]:

	ORIGIN_AIRPORT_ID	DEST_AIRPORT_ID	FL_COUNT	ORIGIN_INDEX	DEST_INDEX
0	10135	10397	77	119	373
1	10135	11433	85	119	1375
2	10135	13930	18	119	3770
3	10140	10397	93	124	373
4	10140	10423	4	124	399
5	10140	10821	64	124	792
6	10140	11259	143	124	1214
7	10140	11292	127	124	1245
8	10140	11298	150	124	1250
9	10140	12191	89	124	2106

From outdegree to weight. Given the outdegree d_i, let $p_{ij} = \frac{1}{d_i}$. In other words, suppose that a random flyer at airport i is *equally likely* to pick any of the destinations directly reachable from i. The following code cell stores that value in a new column, `WEIGHT`.

In [24]:
```
if 'WEIGHT' in segments:
    del segments['WEIGHT']

segments['WEIGHT'] = 1.0 / segments['OUTDEGREE']
display(segments.head(10))

# These should sum to 1.0!
origin_groups = segments[['ORIGIN_INDEX', 'WEIGHT']].groupby('ORIGIN_I
NDEX')
assert np.allclose(origin_groups.sum(), 1.0, atol=10*n_actual*np.finfo
(float).eps), "Rows of $P$ do not sum to 1.0"
```

	ORIGIN_AIRPORT_ID	DEST_AIRPORT_ID	FL_COUNT	ORIGIN_INDEX	DEST_INDEX
0	10135	10397	77	119	373
1	10135	11433	85	119	1375
2	10135	13930	18	119	3770
3	10140	10397	93	124	373
4	10140	10423	4	124	399
5	10140	10821	64	124	792
6	10140	11259	143	124	1214
7	10140	11292	127	124	1245
8	10140	11298	150	124	1250
9	10140	12191	89	124	2106

With my updated `segments` tibble, I constructed a sparse matrix, `P`, corresponding to the state-transition matrix P using SciPy's scipy.sparse.coo_matrix() (https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html) function.

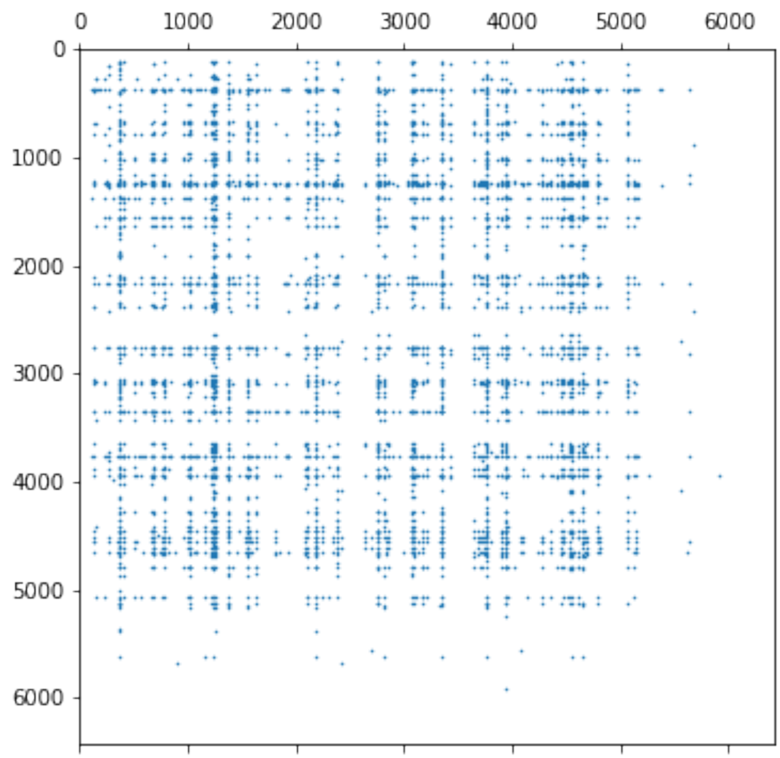
The dimension of the matrix is `n_airports` by `n_airports`. If an airport does not have any outgoing segments in the data, it will appear in the matrix as a row of zeroes.

In [25]:
```
from scipy.sparse import coo_matrix

i = np.array(segments['ORIGIN_INDEX'])
j = np.array(segments['DEST_INDEX'])
v = np.array(segments['WEIGHT'])


P = coo_matrix((v, (i, j)), shape=(n_airports, n_airports))


# Visually inspect my sparse matrix:
spy(P)
```



Computing the steady-state distribution

Armed with the state-transition matrix P, I can now compute the steady-state distribution.

At time $t = 0$, suppose the random flyer is equally likely to be at any airport with an outbound segment, i.e., the flyer is at one of the "actual" origins. I have created a NumPy vector `x0[:]` such that `x0[i]` equals this initial probability of being at airport `i`.

> Note: If some airport i has *no* outbound flights, then $x_i(0) = 0$.

In [26]:

```
'''There are 300 actual origins, so probability that flight starts at one of those aiports
is 1/300 = 0.00333'''
n = P.shape[0]
u = np.ones(n)
row_sums = P.dot(u)
x0 = row_sums
x0 = x0 * 1/300

# Visually inspect my result:
def display_vec_sparsely(x, name='x'):
    i_nz = np.argwhere(x).flatten()
    df_x_nz = pd.DataFrame({'i': i_nz, '{}[i] (non-zero only)'.format(name): x[i_nz]})
    display(df_x_nz.head())
    print("...")
    display(df_x_nz.tail())

display_vec_sparsely(x0, name='x0')
```

	i	x0[i] (non-zero only)
0	119	0.003333
1	124	0.003333
2	125	0.003333
3	130	0.003333
4	138	0.003333

...

	i	x0[i] (non-zero only)
295	5565	0.003333
296	5612	0.003333
297	5630	0.003333
298	5685	0.003333
299	5908	0.003333

Given the state-transition matrix `P`, an initial vector `x0`, and the number of time steps `t_max`, my function `eval_markov_chain(P, x0, t_max)` will compute and return $x(t_{\max})$.

In [27]:
```
def eval_markov_chain(P, x0, t_max):
    P_T = P.T
    x = x0
    for _ in range(t_max):
        x = P_T.dot(x)
    return x


T_MAX = 50
x = eval_markov_chain(P, x0, T_MAX)
display_vec_sparsely(x)

print("\n=== Top 10 airports ===\n")
ranks = np.argsort(-x)
top10 = pd.DataFrame({'Rank': np.arange(1, 11),
                      'Code': airport_codes.iloc[ranks[:10]]['Code'],
                      'Description': airport_codes.iloc[ranks[:10]]['D
escription'],
                      'x(t)': x[ranks[:10]]})
top10[['x(t)', 'Rank', 'Code', 'Description']]
```

	i	x[i] (non-zero only)
0	119	0.000721
1	124	0.005492
2	125	0.000237
3	130	0.000238
4	138	0.000715

```
...
```

	i	x[i] (non-zero only)
295	5565	0.000472
296	5612	0.000239
297	5630	0.001889
298	5685	0.000465
299	5908	0.000239

```
=== Top 10 airports ===
```

```
Out[27]:
```

	x(t)	Rank	Code	Description
373	0.037384	1	10397	Atlanta, GA: Hartsfield-Jackson Atlanta Intern...
3770	0.036042	2	13930	Chicago, IL: Chicago O'Hare International
1245	0.031214	3	11292	Denver, CO: Denver International
3347	0.026761	4	13487	Minneapolis, MN: Minneapolis-St Paul Internati...
2177	0.024809	5	12266	Houston, TX: George Bush Intercontinental/Houston
1250	0.024587	6	11298	Dallas/Fort Worth, TX: Dallas/Fort Worth Inter...
1375	0.024483	7	11433	Detroit, MI: Detroit Metro Wayne County
3941	0.021018	8	14107	Phoenix, AZ: Phoenix Sky Harbor International
4646	0.020037	9	14869	Salt Lake City, UT: Salt Lake City International
1552	0.019544	10	11618	Newark, NJ: Newark Liberty International

Comparing the two rankings. The table below compares my two airport two rankings side-by-side, where the first ranking is the result of the Markov chain analysis and the second ranking is based solely on number of flight segments.

In [29]:
```
top10_with_ranks = top10[['Code', 'Rank', 'Description']].copy()

origins_top10_with_ranks = origins_top10[['ID', 'Description']].copy()
origins_top10_with_ranks.rename(columns={'ID': 'Code'}, inplace=True)
origins_top10_with_ranks['Rank'] = origins_top10.index + 1
origins_top10_with_ranks = origins_top10_with_ranks[['Code', 'Rank', '
Description']]

top10_compare = top10_with_ranks.merge(origins_top10_with_ranks, how='
outer', on='Code',
                                       suffixes=['_MC', '_Seg'])
top10_compare
```

Out[29]:

	Code	Rank_MC	Description_MC	Rank_Seg	Description_Seg
0	10397	1.0	Atlanta, GA: Hartsfield-Jackson Atlanta Intern...	1.0	Atlanta, GA: Hartsfield-Jackson Atlanta Intern...
1	13930	2.0	Chicago, IL: Chicago O'Hare International	2.0	Chicago, IL: Chicago O'Hare International
2	11292	3.0	Denver, CO: Denver International	3.0	Denver, CO: Denver International
3	13487	4.0	Minneapolis, MN: Minneapolis-St Paul Internati...	10.0	Minneapolis, MN: Minneapolis-St Paul Internati...
4	12266	5.0	Houston, TX: George Bush Intercontinental/Houston	NaN	NaN
5	11298	6.0	Dallas/Fort Worth, TX: Dallas/Fort Worth Inter...	6.0	Dallas/Fort Worth, TX: Dallas/Fort Worth Inter...
6	11433	7.0	Detroit, MI: Detroit Metro Wayne County	NaN	NaN
7	14107	8.0	Phoenix, AZ: Phoenix Sky Harbor International	9.0	Phoenix, AZ: Phoenix Sky Harbor International
8	14869	9.0	Salt Lake City, UT: Salt Lake City International	NaN	NaN
9	11618	10.0	Newark, NJ: Newark Liberty International	NaN	NaN
10	12892	NaN	NaN	4.0	Los Angeles, CA: Los Angeles International
11	14771	NaN	NaN	5.0	San Francisco, CA: San Francisco International
12	14747	NaN	NaN	7.0	Seattle, WA: Seattle/Tacoma International
13	12889	NaN	NaN	8.0	Las Vegas, NV: McCarran International

My Markov chain analysis has determined the top 10 airports at which a random flyer ends up, assuming he or she randomly selects directly reachable destinations (left list). While the top three airports are the same in both ranking schemes, there are several notable differences between this ranking as compared to the ranking based instead on historical outbound segments (right list). Note that airports that appear in only one of my top-ten lists are designated as 'NaN' in the other list. I believe that the ranking based on my Markov chain analysis provides a better measure of an airport's importance to the overall airport network.