Course › Midter... › Midter... › Midter...

Midterm 2: Solutions

Problem 0: COVID-19 USA (total value: 11 points)

Version 1.1

This problem is a data cleaning and analysis task that exercises basic pandas, Numpy, and the graph ranking and analysis content of Notebook 11. It consists of five (5) exercises, numbered 0 through 4, worth a total of **11 points.**

- Exercise 0: 3 points
- Exercise 1: 2 points
- Exercise 2: 1 point
- Exercise 3: 3 points
- Exercise 4: 2 points

All exercises are independent, so if you get stuck on one, try moving on to the next one. However, in such cases do look for notes labeled, *"In case Exercise XXX isn't working"*, as you may need to run some code cells that load pre-computed results that will allow you to continue with any subsequent exercises.

Pro-tips.

- If your program behavior seem strange, try resetting the kernel and rerunning everything.
- If you mess up this notebook or just want to start from scratch, save copies of all your partial responses and use `Actions` → `Reset Assignment` to get a fresh, original copy of this notebook. (*Resetting will wipe out any answers you've written so far, so be sure to stash those somewhere safe if you intend to keep or reuse them!*)
- If you generate excessive output (e.g., from an ill-placed `print` statement) that causes the notebook to load slowly or not at all, use `Actions` → `Clear Notebook Output` to get a clean copy. The clean copy will retain your code but remove any generated output. **However**, it will also **rename** the notebook to `clean.xxx.ipynb`. Since the autograder expects a notebook file with the original name, you'll need to rename the clean notebook accordingly.

Good luck!

Background: Transportation networks and infectious disease

One major factor in the spread of infectious diseases like [COVID-19 (https://www.cdc.gov/coronavirus/2019-nCoV/index.html)](https://www.cdc.gov/coronavirus/2019-nCoV/index.html) is the connectivity of our transportation networks. Therefore, let's ask the following question in this problem: to what extent does the connectivity of the airport network help explain in which regions we have seen the most confirmed cases of COVID-19?

We'll focus on the United States network (recall Notebook 11) and analyze data at the level of US states (e.g., Washington state, California, New York state). Our analysis will have three main steps.

1. Let's start by inspecting some recent COVID-19 data on the number of confirmed cases over time, to see which states are seeing the most cases.
2. Next, let's (re-)analyze the airport network to rank the states by their likelihood of seeing air traffic.
3. Finally, we'll compare the state ranking by incidence of COVID-19 with those by airport traffic, to see if there is any "correlation" between the two. We don't expect perfect overlap in these rankings, but if there is substantial overlap, it would provide evidence for the role that air transportation networks play in the spread of the disease.

Before starting, run the code cell below to load some useful functions and packages.

In [1]:
```python
import sys
print(sys.version)

# Needed for loading data:
import pandas as pd
print(f"Pandas version: {pd.__version__}")

# Some problem-specific helper functions:
```

```
# Some problem-specific helper functions:
import problem_utils
from problem_utils import get_path, assert_tibbles_are_equivalent

# For visualization:
from matplotlib.pyplot import figure, plot, semilogy, grid, legend
%matplotlib inline
```

```
3.7.5 (default, Dec 18 2019, 06:24:58)
[GCC 5.5.0 20171010]
Pandas version: 0.25.3
```

Step 1: Inspecting COVID-19 incidence data by state

Researchers at Johns Hopkins University have been tallying the number of confirmed cases of COVID-19 over time. Let's start by assembling the raw data for analysis.

> *Provenance of these data*. JHU made these data are available in this repo on GitHub (https://github.com/CSSEGISandData/COVID-19), but for this problem, we'll use a pre-downloaded copy.

Location of the data. The data are stored in files, one for each day since January 22, 2020. We can use pandas's `read_csv()` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html) to load them into a `DataFrame` object. For example, here is some code to do that for January 22, March 11, and March 22. Take a moment to read this code and observe the output:

In [2]:
```
print("Location of data files:", get_path('covid19/'))
print("Location of Jan 22 data:", get_path('covid19/01-22-2020.csv'))
print("Loading...")
df0 = pd.read_csv(get_path('covid19/01-22-2020.csv'))
print("Done loading. The first 5 rows:")
df0.head(5)
```

```
Location of data files: ./resource/asnlib/publicdata/covid19/
Location of Jan 22 data: ./resource/asnlib/publicdata/covid19/01-22-2020.csv
Loading...
Done loading. The first 5 rows:
```

Out[2]:

	Province/State	Country/Region	Last Update	Confirmed	Deaths	Recovered
0	Anhui	Mainland China	1/22/2020 17:00	1.0	NaN	NaN
1	Beijing	Mainland China	1/22/2020 17:00	14.0	NaN	NaN
2	Chongqing	Mainland China	1/22/2020 17:00	6.0	NaN	NaN
3	Fujian	Mainland China	1/22/2020 17:00	1.0	NaN	NaN
4	Gansu	Mainland China	1/22/2020 17:00	NaN	NaN	NaN

In [3]:
```
df1 = pd.read_csv(get_path('covid19/03-11-2020.csv'))
df1.head(5)
```

Out[3]:

	Province/State	Country/Region	Last Update	Confirmed	Deaths	Recovered	Latitude	Longitude
0	Hubei	China	2020-03-11T10:53:02	67773	3046	49134	30.9756	112.2707
1	NaN	Italy	2020-03-11T21:33:02	12462	827	1045	43.0000	12.0000
2	NaN	Iran	2020-03-11T18:52:03	9000	354	2959	32.0000	53.0000
3	NaN	Korea, South	2020-03-11T21:13:18	7755	60	288	36.0000	128.0000
4	France	France	2020-03-11T22:53:03	2281	48	12	46.2276	2.2137

In [4]:
```
df2 = pd.read_csv(get_path('covid19/03-22-2020.csv'))
df2.head(5)
```

Out[4]:

	FIPS	Admin2	Province_State	Country_Region	Last_Update	Lat	Long_	Confirmed	Deaths	Recovered	Act
0	36061.0	New York City	New York	US	3/22/20 23:45	40.767273	-73.971526	9654	63	0	0
1	36059.0	Nassau	New York	US	3/22/20 23:45	40.740665	-73.589419	1900	4	0	0

2	36119.0	Westchester	New York	US	3/22/20 23:45	41.162784	-73.757417	1873	0	0	0
3	36103.0	Suffolk	New York	US	3/22/20 23:45	40.883201	-72.801217	1034	9	0	0
4	36087.0	Rockland	New York	US	3/22/20 23:45	41.150279	-74.025605	455	1	0	0

Columns. Observe that the column conventions are changing over time, which will make working with this data quite messy if we don't deal with it.

In this problem, we will only be interested in the following four columns:

- `"Province/State"` or `"Province_State"`. If your code encounters the latter (`"Province_State"`), rename it to the former (`"Province/State"`).
- `"Country/Region"` or `"Country_Region"`. Again, rename instances of the latter to the former.
- `"Last Update"` or `"Last_Update"`. Again, rename the latter to the former.
- `"Confirmed"`. This column is named consistently for all the example data.

Missing values. Observe that there may be missing values, which `read_csv()` converts by default to "not-a-number" (`NaN`) values. Recall that these are special floating-point values. As a by-product of using `NaN` values in columns that otherwise contain integers, those integers are *also* converted to floating-point.

Timestamps. Observe that each dataframe has a column named `"Last Update"`, which contain date and time values stored as *strings*. Moreover, they appear to use different formats. Later, we'll want to standardize these, and for that purpose, we'll use pandas's `to_datetime()` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.to_datetime.html) to convert these into Python `datetime` (https://docs.python.org/3/library/datetime.html) objects. That makes them easier to compare (in code) and do simple arithmetic on them (e.g., calculate the number of days in-between). The following code cells demonstrate these features.

In [5]:
```
print(type(df1['Last Update'].loc[0])) # Confirm that these values are strings

# Example: Convert a column to use `datetime` values:
df0['Timestamp'] = pd.to_datetime(df0['Last Update'])
df1['Timestamp'] = pd.to_datetime(df1['Last Update'])
df1.head(5)
```

```
<class 'str'>
```

Out[5]:

	Province/State	Country/Region	Last Update	Confirmed	Deaths	Recovered	Latitude	Longitude	Timestamp
0	Hubei	China	2020-03-11T10:53:02	67773	3046	49134	30.9756	112.2707	2020-03-11 10:53:02
1	NaN	Italy	2020-03-11T21:33:02	12462	827	1045	43.0000	12.0000	2020-03-11 21:33:02
2	NaN	Iran	2020-03-11T18:52:03	9000	354	2959	32.0000	53.0000	2020-03-11 18:52:03
3	NaN	Korea, South	2020-03-11T21:13:18	7755	60	288	36.0000	128.0000	2020-03-11 21:13:18
4	France	France	2020-03-11T22:53:03	2281	48	12	46.2276	2.2137	2020-03-11 22:53:03

In [6]:
```
# Example: Calcuate the difference, in days, between two timestamps
timestamp_0 = df1['Timestamp'].iloc[0]
timestamp_1 = df1['Timestamp'].iloc[1]
delta_t = timestamp_1 - timestamp_0

print(f"* {timestamp_0}  ==> type: {type(timestamp_0)}")
print(f"* {timestamp_1}  ==> type: {type(timestamp_1)}")
print(f"* Difference: ({timestamp_1}) - ({timestamp_0}) == {delta_t}\n  ==> type: {type(delta_t)})")
```

```
* 2020-03-11 10:53:02  ==> type: <class 'pandas._libs.tslibs.timestamps.Timestamp'>
* 2020-03-11 21:33:02  ==> type: <class 'pandas._libs.tslibs.timestamps.Timestamp'>
* Difference: (2020-03-11 21:33:02) - (2020-03-11 10:53:02) == 0 days 10:40:00
  ==> type: <class 'pandas._libs.tslibs.timedeltas.Timedelta'>)
```

You won't need to do date-time arithmetic directly, but standardizing in this way will facilitate things like sorting the data by timestamp.

Getting a list of available data files. Lastly, here is a function to get a list of available daily data files by filename. You don't need to read this code, but do observe the results of the demo call to see how it is useful.

In [7]:
```
def get_covid19_daily_filenames(root=get_path("covid19/")):
```

```
In [7]: def get_covid19_daily_filenames(root=get_path('covid19/')):
    """
    Returns a list of file paths corresponding to JHU's
    daily tallies of COVID-19 cases.
    """
    from os import listdir
    from os.path import isfile
    from re import match

    def covid19_filepath(filebase, root):
        return f"{root}{filebase}"

    def is_covid19_daily_file(filebase, root):
        file_path = covid19_filepath(filebase, root)
        return isfile(file_path) and match('^\d\d-\d\d-2020.csv$', filebase)

    filenames = []
    for b in listdir(root):
        if is_covid19_daily_file(b, root):
            filenames.append(covid19_filepath(b, root))
    return sorted(filenames)

# Demo:
print(repr(get_covid19_daily_filenames()))
```

```
['./resource/asnlib/publicdata/covid19/01-22-2020.csv', './resource/asnlib/publicdata/covid19/01-23-2020.csv', './
resource/asnlib/publicdata/covid19/01-24-2020.csv', './resource/asnlib/publicdata/covid19/01-25-2020.csv', './reso
urce/asnlib/publicdata/covid19/01-26-2020.csv', './resource/asnlib/publicdata/covid19/01-27-2020.csv', './resourc
e/asnlib/publicdata/covid19/01-28-2020.csv', './resource/asnlib/publicdata/covid19/01-29-2020.csv', './resource/as
nlib/publicdata/covid19/01-30-2020.csv', './resource/asnlib/publicdata/covid19/01-31-2020.csv', './resource/asnli
b/publicdata/covid19/02-01-2020.csv', './resource/asnlib/publicdata/covid19/02-02-2020.csv', './resource/asnlib/pu
blicdata/covid19/02-03-2020.csv', './resource/asnlib/publicdata/covid19/02-04-2020.csv', './resource/asnlib/public
data/covid19/02-05-2020.csv', './resource/asnlib/publicdata/covid19/02-06-2020.csv', './resource/asnlib/publicdat
a/covid19/02-07-2020.csv', './resource/asnlib/publicdata/covid19/02-08-2020.csv', './resource/asnlib/publicdata/co
vid19/02-09-2020.csv', './resource/asnlib/publicdata/covid19/02-10-2020.csv', './resource/asnlib/publicdata/covid1
9/02-11-2020.csv', './resource/asnlib/publicdata/covid19/02-12-2020.csv', './resource/asnlib/publicdata/covid19/02
-13-2020.csv', './resource/asnlib/publicdata/covid19/02-14-2020.csv', './resource/asnlib/publicdata/covid19/02-15-
2020.csv', './resource/asnlib/publicdata/covid19/02-16-2020.csv', './resource/asnlib/publicdata/covid19/02-17-202
0.csv', './resource/asnlib/publicdata/covid19/02-18-2020.csv', './resource/asnlib/publicdata/covid19/02-19-2020.cs
v', './resource/asnlib/publicdata/covid19/02-20-2020.csv', './resource/asnlib/publicdata/covid19/02-21-2020.csv',
'./resource/asnlib/publicdata/covid19/02-22-2020.csv', './resource/asnlib/publicdata/covid19/02-23-2020.csv', './r
esource/asnlib/publicdata/covid19/02-24-2020.csv', './resource/asnlib/publicdata/covid19/02-25-2020.csv', './resou
rce/asnlib/publicdata/covid19/02-26-2020.csv', './resource/asnlib/publicdata/covid19/02-27-2020.csv', './resource/
asnlib/publicdata/covid19/02-28-2020.csv', './resource/asnlib/publicdata/covid19/02-29-2020.csv', './resource/asnl
ib/publicdata/covid19/03-01-2020.csv', './resource/asnlib/publicdata/covid19/03-02-2020.csv', './resource/asnlib/p
ublicdata/covid19/03-03-2020.csv', './resource/asnlib/publicdata/covid19/03-04-2020.csv', './resource/asnlib/publi
cdata/covid19/03-05-2020.csv', './resource/asnlib/publicdata/covid19/03-06-2020.csv', './resource/asnlib/publicdat
a/covid19/03-07-2020.csv', './resource/asnlib/publicdata/covid19/03-08-2020.csv', './resource/asnlib/publicdata/co
vid19/03-09-2020.csv', './resource/asnlib/publicdata/covid19/03-10-2020.csv', './resource/asnlib/publicdata/covid1
9/03-11-2020.csv', './resource/asnlib/publicdata/covid19/03-12-2020.csv', './resource/asnlib/publicdata/covid19/03
-13-2020.csv', './resource/asnlib/publicdata/covid19/03-14-2020.csv', './resource/asnlib/publicdata/covid19/03-15-
2020.csv', './resource/asnlib/publicdata/covid19/03-16-2020.csv', './resource/asnlib/publicdata/covid19/03-17-202
0.csv', './resource/asnlib/publicdata/covid19/03-18-2020.csv', './resource/asnlib/publicdata/covid19/03-19-2020.cs
v', './resource/asnlib/publicdata/covid19/03-20-2020.csv', './resource/asnlib/publicdata/covid19/03-21-2020.csv',
'./resource/asnlib/publicdata/covid19/03-22-2020.csv', './resource/asnlib/publicdata/covid19/03-23-2020.csv', './r
esource/asnlib/publicdata/covid19/03-24-2020.csv']
```

Exercise 0 (3 points): Data loading and cleaning

Given `filenames`, a list of filenames that might be generated by `get_covid19_filenames()` above, complete the function `load_covid19_daily_data(filenames)` below so that it reads all of this data and combines it into a single tibble (as a pandas `DataFrame`) containing only the following columns:

- `"Province/State"`: Same contents as the original data frames.
- `"Country/Region"`: Same contents as the original data frames.
- `"Confirmed"`: Same contents as the original data frames.
- `"Timestamp"`: The values from the `"Last Update"` columns, but **converted** to `datetime` objects per the demonstration discussed previously.

In addition, your code should do the following:

1. Don't forget that sometimes `"Province/State"`, `"Country/Region"`, and `"Last Update"` are written differently, so be sure to handle those cases.
2. If there are any duplicate rows (i.e., two or more rows whose values are identical), only one of the rows should be retained.
3. In the `"Confirmed"` column, any missing values should be replaced by (0). Also, this column should be converted to have an integer type. (*Hint:* Consider `Series.fillna()` and `Series.astype()`.)
4. Your code should *not* depend on the input files having any specific columns other than the ones directly relevant to producing the above output, i.e., `"Province/State"`, `"Country/Region"`, `"Confirmed"`, and `"Last Update"`. It should also not depend on any particular ordering of the columns.

Hint 0. Per the preceding examples, use `pd.read_csv()` to read the contents of each file into a data frame. However, the `filenames` list will already include a valid path, so you do **not** need to use `get_path()`.

Hint 1. Recall that you can use `pd.concat()` to concatenate data frames; one tweak in here is to use its `ignore_index=True` parameter to get a clean tibble-like index.

Hint 2. To easily drop duplicate rows, look for a relevant pandas built-in function.

In [8]:
```
def load_covid19_daily_data(filenames):
    ### BEGIN SOLUTION
    from pandas import read_csv, concat, to_datetime
    df_list = []
    for filename in filenames:
        df = read_csv(filename).rename(columns={"Province_State": "Province/State",
                                                "Country_Region": "Country/Region",
                                                "Last_Update": "Last Update"})
        df = df[["Province/State", "Country/Region", "Confirmed", "Last Update"]]
        df["Last Update"] = to_datetime(df["Last Update"])
        df['Confirmed'] = df['Confirmed'].fillna(0).astype(int)
        df_list.append(df)
    df_combined = concat(df_list)
    df_combined.rename(columns={"Last Update": "Timestamp"}, inplace=True)
    df_combined.drop_duplicates(inplace=True)
    return df_combined.reset_index(drop=True)
    ### END SOLUTION
```

In [9]:
```
# Demo of your function:
df = load_covid19_daily_data(get_covid19_daily_filenames())

print(f"There are {len(df)} rows in your data frame.")
print("The first five are:")
display(df.head(5))

print("A random sample of five additional rows:")
df.sample(5).sort_index()
```

```
There are 7588 rows in your data frame.
The first five are:
```

	Province/State	Country/Region	Confirmed	Timestamp
0	Anhui	Mainland China	1	2020-01-22 17:00:00
1	Beijing	Mainland China	14	2020-01-22 17:00:00
2	Chongqing	Mainland China	6	2020-01-22 17:00:00
3	Fujian	Mainland China	1	2020-01-22 17:00:00
4	Gansu	Mainland China	0	2020-01-22 17:00:00

```
A random sample of five additional rows:
```

Out[9]:

	Province/State	Country/Region	Confirmed	Timestamp
865	Beijing	Mainland China	326	2020-02-09 03:43:02
879	Jilin	Mainland China	78	2020-02-09 09:03:04
1908	Unassigned Location (From Diamond Princess)	US	45	2020-03-02 19:53:03
2716	NaN	Switzerland	491	2020-03-10 23:53:02
4583	NaN	The Bahamas	0	2020-03-19 12:13:38

In [10]:
```
# Test cell: `ex0__load_covid19_daily_data` (3 points)

### BEGIN HIDDEN TESTS
def ex0_soln(filenames):
    from pandas import read_csv, concat, to_datetime
    df_combined = None
    for filename in filenames:
        df = read_csv(filename).rename(columns={"Province_State": "Province/State",
                                                "Country_Region": "Country/Region",
                                                "Last_Update": "Last Update"})
        df = df[["Province/State", "Country/Region", "Confirmed", "Last Update"]]
        df["Timestamp"] = to_datetime(df["Last Update"])
```

```
        del df["Last Update"]
        df['Confirmed'] = df['Confirmed'].fillna(0).astype(int)
        if df_combined is None:
            df_combined = df
        else:
            df_combined = concat([df_combined, df], ignore_index=True)
    df_combined.drop_duplicates(inplace=True)
    return df_combined.reset_index(drop=True)

#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#

def ex0_gen_soln(soln_file=get_path('covid19/ex0_soln.csv'), force=False):
    from os.path import isfile
    if isfile(soln_file) and not force:
        print(f"Solution file, '{soln_file}', already exists. NOT regenerating...")
    else:
        print(f"Generating solution file, '{soln_file}'...")
        df = ex0_soln(get_covid19_daily_filenames())
        df.to_csv(soln_file, index=False)
        print("==> Done!")

def ex0_gen_locales(force=False):
    from os.path import isfile
    from json import dump
    from collections import defaultdict
    from pandas import isna
    outfilename = get_path("locales.json")
    if isfile(outfilename) and not force:
        print(f"Locales file, '{outfilename}', exists; skipping generation...")
        return
    print(f"Generating locales file, '{outfilename}'...")
    df = load_covid19_daily_data(get_covid19_daily_filenames())
    locales = defaultdict(set)
    for k, row in df.iterrows():
        country = row["Country/Region"]
        province = row["Province/State"]
        if not (isna(country) or isna(province)):
            locales[country] |= {province}
    for country in locales:
        locales[country] = list(locales[country])
    with open(get_path("locales.json"), "wt") as fp:
        dump(locales, fp, indent=2, sort_keys=True)
    print("==> Done generating locales file.")

ex0_gen_locales()
ex0_gen_soln(force=False)
### END HIDDEN TESTS

def ex0_random_value():
    from random import random, randint, choice
    from numpy import nan
    from problem_utils import ex0_random_date, ex0_random_string
    options = [randint(-100, 100) # int
               , ex0_random_string(randint(1, 10)) # string
               , ex0_random_date(), # date
               '', # implicit NaN
               nan # explicit NaN
              ]
    return choice(options)

def ex0_get_locales(filename=get_path("locales.json")):
    from json import load
    with open(filename, "rt") as fp:
        locales = load(fp)
```

```
        locales = load(fp)
    return locales

def ex0_gen_row(locales, num_dummies=0):
    from datetime import datetime
    from random import choice, random, randint
    from numpy import nan
    from problem_utils import ex0_random_date
    country = choice(list(locales.keys()))
    province = nan if random() <= 0.1 else choice(locales[country])
    confirmed = 0 if random() <= 0.1 else randint(1, 100000)
    last_updated = ex0_random_date()
    if num_dummies:
        dummy_vals = tuple([ex0_random_value() for _ in range(num_dummies)])
    else:
        dummy_vals = ()
    return (country, province, confirmed, last_updated, *dummy_vals)

def ex0_gen_df():
    from random import randint, random
    from pandas import DataFrame
    from problem_utils import ex0_random_string

    locales = ex0_get_locales()

    # Generate random columns, which the student should ignore
    num_dummy_cols = randint(1, 4)
    dummy_cols = []
    while len(dummy_cols) != num_dummy_cols:
        dummy_cols = list({ex0_random_string(5) for _ in range(num_dummy_cols)})

    # Generate a bunch of random rows
    num_trials = randint(10, 50)
    rows = [ex0_gen_row(locales, num_dummy_cols) for _ in range(num_trials)]

    # Remove any initial duplicates
    rows = sorted(rows, key=lambda x: repr(x))
    rows_soln = [rows[0]]
    for r in rows[1:]:
        if repr(r) != repr(rows_soln[-1]):
            rows_soln.append(r)

    # Construct the solution tibble
    cols_in = ["Country/Region" if random() < 0.75 else "Country_Region",
               "Province/State" if random() < 0.75 else "Province_State",
               "Confirmed",
               "Last Update" if random() < 0.75 else "Last_Update"]
    cols_out = ["Country/Region", "Province/State", "Confirmed", "Last Update"]
    df_soln = DataFrame(rows_soln, columns=cols_out + dummy_cols)[cols_out] \
              .rename(columns={"Last Update": "Timestamp"})

    # Generate a corresponding input tibble
    rows_in = []
    for r in rows_soln:
        s = list(r)
        if s[2] == 0:
            s[2] = '' # NaN counts
        r_in = tuple(s)
        rows_in.append(r_in)
        if random() <= 0.15: # Random duplicates
            for _ in range(randint(1, 4)):
                rows_in.append(r_in)
    df_in = DataFrame(rows_in, columns=cols_in + dummy_cols)

    return df_in, df_soln

def ex0_split_df(df, max_splits=5):
    from random import randint
    from numpy import arange, sort, append
    from numpy.random import shuffle, choice
    # Shuffle the rows
    df = df.sample(frac=1).reset_index(drop=True)

    # Split the rows
    df_split = []
    num_splits = min(randint(0, max_splits), len(df))
    if num_splits > 0:
        split_inds = sort(choice(arange(len(df)), size=num_splits, replace=False))
        if split_inds[0] > 0:
            split_inds = append(0, split_inds)
        if split_inds[-1] < len(df):
```

```
            split_inds = append(split_inds, len(df))
        for i, j in zip(split_inds[:-1], split_inds[1:]):
            df_ij = df.iloc[i:j].reset_index(drop=True)
            df_split.append(df_ij)

    return df_split if num_splits else [df]

def ex0_certify_metadata(df):
    df_cols = set(df.columns)
    true_cols = {"Province/State", "Country/Region", "Confirmed", "Timestamp"}
    too_many_cols = df_cols - true_cols
    assert not too_many_cols, f"*** You have too many columns, including {too_many_cols}. ***"
    missing_cols = true_cols - df_cols
    assert not missing_cols, f"*** You are missing some columns, namely, {missing_cols}. ***"

    from pandas.api.types import is_integer_dtype
    assert is_integer_dtype(df["Confirmed"]), \
           '*** `"Confirmed"` column has a non-integer type ({type(df["Confirmed"])}). ***'

    from numpy import datetime64
    from pandas import Index
    assert df.select_dtypes(include=datetime64).columns == Index(["Timestamp"]), \
           '*** Your data frame must have a "Timestamp" column containing `datetime` values.'

def ex0_check():
    from problem_utils import canonicalize_tibble, tibbles_left_matches_right
    from os import remove
    from os.path import isfile
    print("Generating synthetic input files...")
    df_in, df_soln = ex0_gen_df()
    df_split = ex0_split_df(df_in)
    filenames = []
    for k, df_k in enumerate(df_split):
        filenames.append(f'./ex0_df{k}.csv')
        print(f"- {filenames[-1]}")
        df_k.to_csv(filenames[-1], index=False)
    try:
        print("Testing your solution...")
        df = load_covid19_daily_data(filenames)
        ex0_certify_metadata(df)
        assert tibbles_left_matches_right(df, df_soln, verbose=True), \
               "*** Your computed solution does not match ours. ***"
    except:
        print("\n=== Expected solution ===")
        display(canonicalize_tibble(df_soln, remove_index=True))
        print("\n=== Your computed solution ===")
        display(canonicalize_tibble(df, remove_index=True))
        print(f"\nNOTE: To see the original input files, inspect {filenames}.")
        raise
    else:
        print("Cleaning up input files...")
        for f in filenames:
            if isfile(f):
                print(f"- {f}")
                remove(f)


for trial in range(5):
    print(f"========== Trial #{trial} ==========")
    ex0_check()

print("\n(Passed.)")
```

```
Locales file, './resource/asnlib/publicdata/locales.json', exists; skipping generation...
Solution file, './resource/asnlib/publicdata/covid19/ex0_soln.csv', already exists. NOT regenerating...
========== Trial #0 ==========
Generating synthetic input files...
- ./ex0_df0.csv
- ./ex0_df1.csv
- ./ex0_df2.csv
- ./ex0_df3.csv
Testing your solution...
Cleaning up input files...
- ./ex0_df0.csv
- ./ex0_df1.csv
- ./ex0_df2.csv
- ./ex0_df3.csv
========== Trial #1 ==========
Generating synthetic input files...
- ./ex0_df0.csv
```

```
- ./ex0_df1.csv
Testing your solution...
Cleaning up input files...
- ./ex0_df0.csv
- ./ex0_df1.csv
========== Trial #2 ==========
Generating synthetic input files...
- ./ex0_df0.csv
- ./ex0_df1.csv
- ./ex0_df2.csv
Testing your solution...
Cleaning up input files...
- ./ex0_df0.csv
- ./ex0_df1.csv
- ./ex0_df2.csv
========== Trial #3 ==========
Generating synthetic input files...
- ./ex0_df0.csv
- ./ex0_df1.csv
- ./ex0_df2.csv
- ./ex0_df3.csv
Testing your solution...
Cleaning up input files...
- ./ex0_df0.csv
- ./ex0_df1.csv
- ./ex0_df2.csv
- ./ex0_df3.csv
========== Trial #4 ==========
Generating synthetic input files...
- ./ex0_df0.csv
- ./ex0_df1.csv
- ./ex0_df2.csv
Testing your solution...
Cleaning up input files...
- ./ex0_df0.csv
- ./ex0_df1.csv
- ./ex0_df2.csv

(Passed.)
```

A combined data frame

Whether you solved Exercise 0 or not, we have prepared a file of pre-cleaned and combined COVID-19 data. Below, the variable `df_covid19` holds these data. You will need it in the subsequent exercises, so be sure to run this cell and do not modify the variable!

In [11]:
```
df_covid19 = pd.read_csv(get_path('covid19/ex0_soln.csv'), parse_dates=["Timestamp"])
df_covid19 = df_covid19.groupby(["Province/State", "Country/Region", "Timestamp"], as_index=False).sum()
             # ^^^ Above `.groupby()` needed because of a change in US reporting on March 22, 2020
df_covid19.sample(5)
```

Out[11]:

	Province/State	Country/Region	Timestamp	Confirmed
1220	Hubei	China	2020-03-18 12:13:09	67800
1084	Heilongjiang	Mainland China	2020-02-29 12:03:07	480
16	Alabama	US	2020-03-23 23:19:34	173
1149	Hillsborough, FL	US	2020-03-03 18:33:02	2
246	British Columbia	Canada	2020-03-11 20:00:00	64

US state-by-state data

The dataset includes confirmed cases in the US. For instance, run this cell to see a sample of the US rows.

In [12]:
```
is_us = (df_covid19["Country/Region"] == "US")
df_covid19[is_us].sample(5)
```

Out[12]:

	Province/State	Country/Region	Timestamp	Confirmed
1845	Michigan	US	2020-03-16 14:38:46	53
1727	Louisiana	US	2020-03-15 18:20:19	91
2388	San Diego County, CA	US	2020-02-21 05:43:02	2

2388	San Diego County, CA	US	2020-02-21 05:43:02	2
2950	US	US	2020-03-21 19:43:03	1
1897	Missouri	US	2020-03-24 23:37:31	226

You should see some cases where the `"Province/State"` field is exactly the name of a US state, like `"Georgia"` or `"California"`, and other cases where you might see a more or less detailed location (e.g., a city name and a statee, like `"Middlesex County, MA"`).

For subsequent analysis, we will only be interested in the rows containing **state names**. For instance, here are all the rows associated with `"Georgia"`.

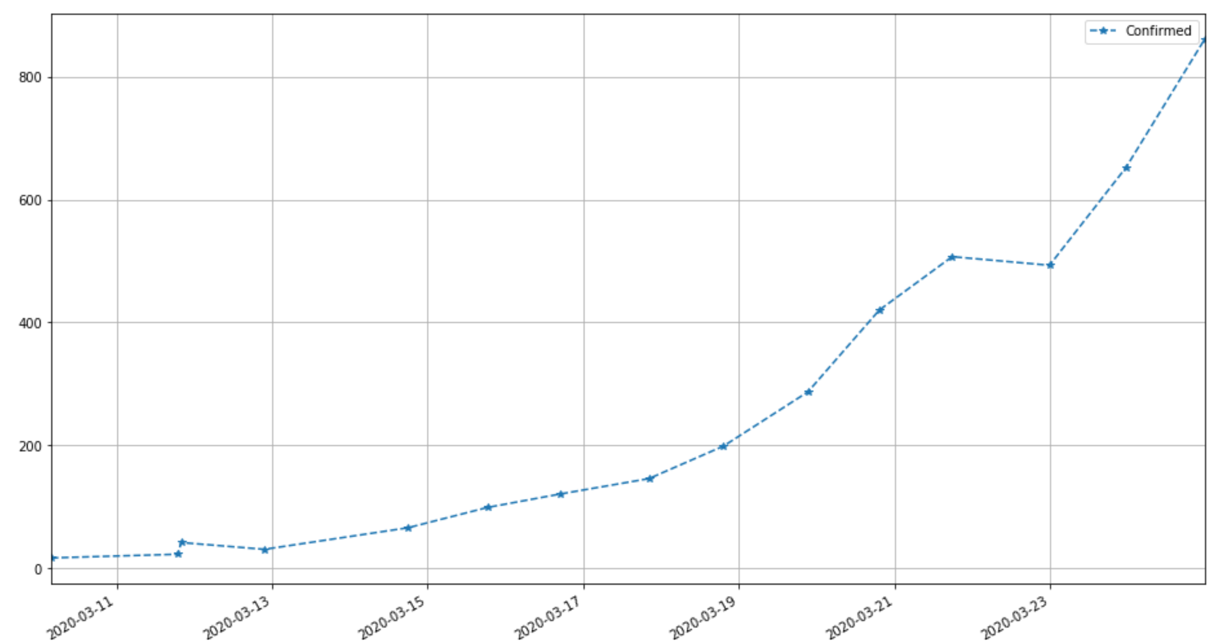
In [13]:
```
is_georgia = (df_covid19["Province/State"] == "Georgia")
df_covid19[is_us & is_georgia]
```

Out[13]:

	Province/State	Country/Region	Timestamp	Confirmed
676	Georgia	US	2020-03-10 03:53:03	17
677	Georgia	US	2020-03-11 18:52:03	23
678	Georgia	US	2020-03-11 20:00:00	42
679	Georgia	US	2020-03-12 21:39:10	31
680	Georgia	US	2020-03-14 17:53:03	66
681	Georgia	US	2020-03-15 18:20:19	99
682	Georgia	US	2020-03-16 16:53:06	121
683	Georgia	US	2020-03-17 20:13:22	146
684	Georgia	US	2020-03-18 19:14:34	199
685	Georgia	US	2020-03-19 21:13:35	287
686	Georgia	US	2020-03-20 19:13:30	420
687	Georgia	US	2020-03-21 17:43:03	507
688	Georgia	US	2020-03-22 23:45:00	493
689	Georgia	US	2020-03-23 23:19:34	652
690	Georgia	US	2020-03-24 23:37:31	861

Given these data, we can order by timestamp and plot confirmed cases over time.

In [14]:
```
df_covid19[is_us & is_georgia] \
    .sort_values(by="Timestamp") \
    .plot(x="Timestamp", y="Confirmed", figsize=(16, 9), style='*--')
grid()
```



Timestamp

Exercise 1 (2 points): US state-by-state data

Complete the function, `get_us_states(df)`, below, where

- its input, `df`, is a data frame structured like the combined COVID-19 data frame (`df_covid19`), having the columns `"Province/State"`, `"Country/Region"`, `"Confirmed"`, `"Timestamp"`;
- and it returns a tibble containing only those rows of `df` that are from the United States where the `"Province/State"` field is exactly the name of any one of the US states.

Regarding the second requirement, the returned object should include a row where the `"Province/State"` field is `"Georgia"`, but it should **not** include a row where this field is, say, `"Atlanta, GA"`. (Put differently, we will assume the state-level accounts already include city-level counts.)

The tibble returned by your function should only have these three columns:

1. `"Confirmed"`: The number of confirmed cases, taken from the input `df`.
2. `"Timestamp"`: The timestamp taken from the input `df`.
3. `"ST"`: The two-letter **abbreviation** for the state's name.

Pay attention to item (3): your returned tibble should not have the state's full name, but rather, its two-letter postal code abbreviation (e.g., `"GA"` instead of `"Georgia"`). To help you out, here is a code cell that defines a data frame called `STATE_NAMES` that holds both a list of state names and their two-letter abbreviations.

> **Note**: The test cell for this exercise reuses functions defined in the test cell for Exercise 0. So even if you skipped Exercise 0, please run its test cell before running the one below.

In [15]:
```
STATE_NAMES = pd.read_csv(get_path('us_states.csv'))
print(f"There are {len(STATE_NAMES)} US states. The first and last three, along with their two-letter postal code a
bbreviations, are as follows (in alphabetical order):")
display(STATE_NAMES.head(3))
print("...")
display(STATE_NAMES.tail(3))
```

```
There are 50 US states. The first and last three, along with their two-letter postal code abbreviations, are as fo
llows (in alphabetical order):
```

	Name	Abbrv
0	Alabama	AL
1	Alaska	AK
2	Arizona	AZ

...

	Name	Abbrv
47	West Virginia	WV
48	Wisconsin	WI
49	Wyoming	WY

In [16]:
```
def get_us_states(df):
    ### BEGIN SOLUTION
    return get_us_states__0(df)

# Solution 0: `.merge()`
def get_us_states__0(df):
    df_state_names = STATE_NAMES.rename(columns={"Name": "Province/State"})
    df_states = df_state_names.merge(df)
    del df_states["Province/State"]
    del df_states["Country/Region"]
    return df_states.rename(columns={"Abbrv": "ST"})

# Solution 1: `.isin() / .str.replace()`
def get_us_states__1(df):
    is_us = (df["Country/Region"] == "US")
    is_state = df[is_us]["Province/State"].isin(STATE_NAMES['Name'])
    df_state = df[is_us & is_state]
    df_st = df_state.rename(columns={"Province/State": "ST"})
```

```
    for _, row in STATE_NAMES.iterrows():
        pattern = f'^{row["Name"]}$'
        replacement = row["Abbrv"]
        df_st["ST"] = df_st["ST"].str.replace(pattern, replacement)
    del df_st["Country/Region"]
    return df_st

# Solution 2: `.isin()` / `.map()`
def get_us_states__2(df):
    is_us = df['Country/Region'] == 'US'
    names = STATE_NAMES["Name"]
    is_us_state = is_us & df['Province/State'].isin(names)
    abbrvs = STATE_NAMES["Abbrv"]
    name2abbrv = {name: st for name, st in zip(names, abbrvs)}
    df_us = df[is_us_state].copy()
    df_us['ST'] = df_us['Province/State'].map(name2abbrv)
    del df_us["Province/State"]
    del df_us["Country/Region"]
    return df_us
    ### END SOLUTION
```

In [17]:
```
# Test cell: `ex1__get_us_states` (2 points)

### BEGIN HIDDEN TESTS
def ex1_gen_soln(force=False):
    def ex1_soln(df):
        df_state_names = STATE_NAMES.rename(columns={"Name": "Province/State"})
        df_states = df_state_names.merge(df)
        del df_states["Province/State"]
        del df_states["Country/Region"]
        return df_states.rename(columns={"Abbrv": "ST"})
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    from os.path import isfile
    soln_file = get_path('covid19/ex1_soln.csv')
    if isfile(soln_file) and not force:
        print(f"Solution file, '{soln_file}', already exists. NOT regenerating...")
    else:
        print(f"Generating solution file, '{soln_file}'...")
        df = ex1_soln(df_covid19)
        df.to_csv(soln_file, index=False)
        print("==> Done!")

ex1_gen_soln(force=False)
### END HIDDEN TESTS

def ex1_gen_row(states):
    from datetime import datetime
    from random import random, randint, choice
    from problem_utils import ex0_random_date, ex0_random_string
    def rand_str(): return ex0_random_string(randint(1, 10))

    confirmed = randint(1, 10000)
    timestamp = ex0_random_date()

    # Choose a province
    locales = ex0_get_locales()
    p = random()
    if p < 0.5: # Non US country
        country = choice(list(set(locales.keys()) - {"US"}))
        province = choice(locales[country])
        is_state = False
    else:
        country = "US"
```

```
        country = "US"
        if p < 0.75:
            non_states = set(locales["US"]) - set(states["Name"])
            province = choice(list(non_states))
            is_state = False
        else:
            province = choice(states["Name"])
            is_state = True
    return timestamp, confirmed, country, province, is_state

def ex1_gen_df(max_rows, states):
    from random import randint
    from pandas import DataFrame, concat
    st_lookup = states.set_index("Name")
    num_rows = randint(1, max_rows)
    df_list = []
    cols_df = ["Timestamp", "Confirmed", "Country/Region", "Province/State"]
    df_soln_list = []
    cols_df_soln = ["Timestamp", "Confirmed", "ST"]
    for _ in range(num_rows):
        ts, conf, country, province, is_state = ex1_gen_row(states)
        df0 = DataFrame([[ts, conf, country, province]], columns=cols_df)
        df_list.append(df0)
        if is_state:
            st = st_lookup.loc[province]["Abbrv"]
            df0_soln = DataFrame([[ts, conf, st]], columns=cols_df_soln)
            df_soln_list.append(df0_soln)
    assert len(df_list) > 0, "*** Problem with the test cell! ***"
    df = concat(df_list, ignore_index=True).sample(frac=1).reset_index(drop=True)
    if len(df_soln_list) == 0:
        df_soln = DataFrame(columns=cols_df_soln)
    else:
        df_soln = concat(df_soln_list, ignore_index=True).sample(frac=1).reset_index(drop=True)
    return df, df_soln

def ex1_check():
    df, df_soln = ex1_gen_df(20, STATE_NAMES)
    try:
        df_your_soln = get_us_states(df)
        assert_tibbles_are_equivalent(df_soln, df_your_soln)
    except:
        print("\n*** ERROR DETECTED ***")
        print("Input data frame:")
        display(df)
        print("Expected solution:")
        display(df_soln)
        print("Your solution:")
        display(df_your_soln)
        raise

for trial in range(10):
    print(f"=== Trial #{trial} / 9 ===")
    ex1_check()

print("\n(Passed.)")
```

```
Solution file, './resource/asnlib/publicdata/covid19/ex1_soln.csv', already exists. NOT regenerating...
=== Trial #0 / 9 ===
=== Trial #1 / 9 ===
=== Trial #2 / 9 ===
=== Trial #3 / 9 ===
=== Trial #4 / 9 ===
=== Trial #5 / 9 ===
=== Trial #6 / 9 ===
=== Trial #7 / 9 ===
=== Trial #8 / 9 ===
=== Trial #9 / 9 ===

(Passed.)
```

US state-by-state data

Whether your Exercise 1 is working or not, please run the following code cell. It loads a pre-generated data frame containing just the state-level COVID-19 confirmed cases data into a variable named, `df_covid19_us`. You will need it in the subsequent exercises, so do not modify it!

In [18]:
```
df_covid19_us = pd.read_csv(get_path('covid19/ex1_soln.csv'), parse_dates=["Timestamp"])
df_covid19_us.sample(5).sort_values(by=["ST", "Timestamp"])
```

Out[18]:

	ST	Timestamp	Confirmed
244	KY	2020-03-19 23:43:04	37
371	MT	2020-03-22 23:45:00	22
594	TX	2020-03-16 23:53:03	85
609	UT	2020-03-16 22:33:03	39
618	VT	2020-03-11 20:00:00	2

Exercise 2 (1 point): Ranking by confirmed cases

Let `df` be a data frame like `df_covid19_us`, which would be produced by a correctly functioning `get_us_states()` (Exercise 1). Complete the function `rank_states_by_cases(df)` so that it returns a **Python `list`** of states in decreasing order of the **maximum** number of confirmed cases in that state.

In [19]:
```
def rank_states_by_cases(df):
    ### BEGIN SOLUTION
    return rank_states_by_cases__0(df)

# Method 0
def rank_states_by_cases__0(df):
    return df.groupby("ST").max().sort_values(by="Confirmed", ascending=False).index.tolist()

# Method 1
def rank_states_by_cases__1(df):
    max_values = []
    for st in STATE_NAMES["Abbrv"]:
        df_st = df[df["ST"] == st]
        v = df_st["Confirmed"].max()
        max_values.append((st, v))
    return [st for st, v in sorted(max_values, key=lambda x: x[1], reverse=True)]
    ### END SOLUTION

your_covid19_rankings = rank_states_by_cases(df_covid19_us)
assert isinstance(your_covid19_rankings, list), "Did you return a Python `list` as instructed?"
print(f"Your computed ranking:\n==> {repr(your_covid19_rankings)}\n")
```

```
Your computed ranking:
==> ['NY', 'NJ', 'CA', 'WA', 'MI', 'IL', 'LA', 'FL', 'MA', 'PA', 'GA', 'TX', 'TN', 'CO', 'CT', 'OH', 'WI', 'NC',
'AZ', 'MD', 'UT', 'SC', 'NV', 'IN', 'VA', 'MO', 'AL', 'MN', 'OR', 'MS', 'AR', 'KY', 'RI', 'ME', 'DE', 'NH', 'OK',
'HI', 'NM', 'VT', 'IA', 'KS', 'ID', 'NE', 'MT', 'AK', 'ND', 'WY', 'SD', 'WV']
```

In [20]:
```
df_covid19_us.head()
```

Out[20]:

	ST	Timestamp	Confirmed
0	AL	2020-03-11 20:00:00	5
1	AL	2020-03-14 16:53:03	6
2	AL	2020-03-15 18:20:19	12
3	AL	2020-03-16 22:33:03	29
4	AL	2020-03-17 23:13:10	39

In [21]:
```
# Test cell: `ex2__rank_states_by_cases` (1 point)

### BEGIN HIDDEN TESTS
def ex2_gen_soln(force=False):
    def ex2_soln(df):
        return df.groupby("ST").max().sort_values(by="Confirmed", ascending=False).index.tolist()
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
```

```
    #
    #
    #
    #
    #
    from os.path import isfile
    soln_file = get_path('covid19/ex2_soln.txt')
    if isfile(soln_file) and not force:
        print(f"Solution file, '{soln_file}', already exists. NOT regenerating...")
    else:
        print(f"Generating solution file, '{soln_file}'...")
        results = ex2_soln(df_covid19_us)
        with open(soln_file, "wt") as fp:
            for st in results:
                fp.write(f"{st}\n")
        print("==> Done!")

ex2_gen_soln(force=False)
### END HIDDEN TESTS

def ex2_gen_df(st):
    from problem_utils import ex0_random_date
    from random import randint
    from pandas import DataFrame
    num_rows = randint(1, 5)
    confs = []
    tss = []
    max_conf = -1
    for k in range(num_rows):
        confs.append(randint(1, 1000))
        if confs[-1] > max_conf: max_conf = confs[-1]
        tss.append(ex0_random_date())
    df_st = DataFrame({"ST": [st] * num_rows, "Confirmed": confs, "Timestamp": tss})
    return df_st, max_conf

def ex2_check():
    from random import randint, sample
    from pandas import concat
    num_states = randint(1, 5)
    states = sample(list(STATE_NAMES["Abbrv"]), num_states)
    vals = []
    df_list = []
    for st in states:
        df_st, max_conf = ex2_gen_df(st)
        df_list.append(df_st)
        vals.append((st, max_conf))
    df = concat(df_list, ignore_index=True).sort_values(by="Timestamp").reset_index(drop=True)
    soln = [s for s, v in sorted(vals, key=lambda x: x[1], reverse=True)]
    try:
        your_soln = rank_states_by_cases(df)
        assert len(soln) == len(your_soln), \
               f"*** Your solution has {len(your_soln)} entries instead of {len(soln)} ***"
        assert all([a == b for a, b in zip(soln, your_soln)]), \
               f"*** Solutions do not match ***"
    except:
        print("\n*** ERROR CASE ***\n")
        print("Input:")
        display(df)
        print("Expected solution:")
        display(soln)
        print("Your solution:")
        display(your_soln)
        raise

for trial in range(10):
    print(f"=== Trial #{trial} / 9 ===")
    ex2_check()

print("\n(Passed.)")
```

```
Solution file, './resource/asnlib/publicdata/covid19/ex2_soln.txt', already exists. NOT regenerating...
=== Trial #0 / 9 ===
=== Trial #1 / 9 ===
=== Trial #2 / 9 ===
=== Trial #3 / 9 ===
=== Trial #4 / 9 ===
=== Trial #5 / 9 ===
=== Trial #6 / 9 ===
=== Trial #7 / 9 ===
=== Trial #8 / 9 ===
```

```
=== Trial #9 / 9 ===

(Passed.)
```

(In case Exercise 2 isn't working) Ranking by confirmed cases

In case you can't get a working solution to Exercise 2, we have prepared a ranked list of states by confirmed cases. The code cell below reads this list and stores it in the variable, `covid19_rankings`. You will need it in the subsequent exercises, so do not modify it!

In [22]:
```python
with open(get_path('covid19/ex2_soln.txt'), "rt") as fp:
    covid19_rankings = [s.strip() for s in fp.readlines()]
print(repr(covid19_rankings))
```

```
['NY', 'NJ', 'CA', 'WA', 'MI', 'IL', 'LA', 'FL', 'MA', 'PA', 'GA', 'TX', 'TN', 'CO', 'CT', 'OH', 'WI', 'NC', 'AZ',
'MD', 'UT', 'SC', 'NV', 'IN', 'VA', 'MO', 'AL', 'MN', 'OR', 'MS', 'AR', 'KY', 'RI', 'ME', 'DE', 'NH', 'OK', 'HI',
'NM', 'VT', 'IA', 'KS', 'ID', 'NE', 'MT', 'AK', 'ND', 'WY', 'SD', 'WV']
```

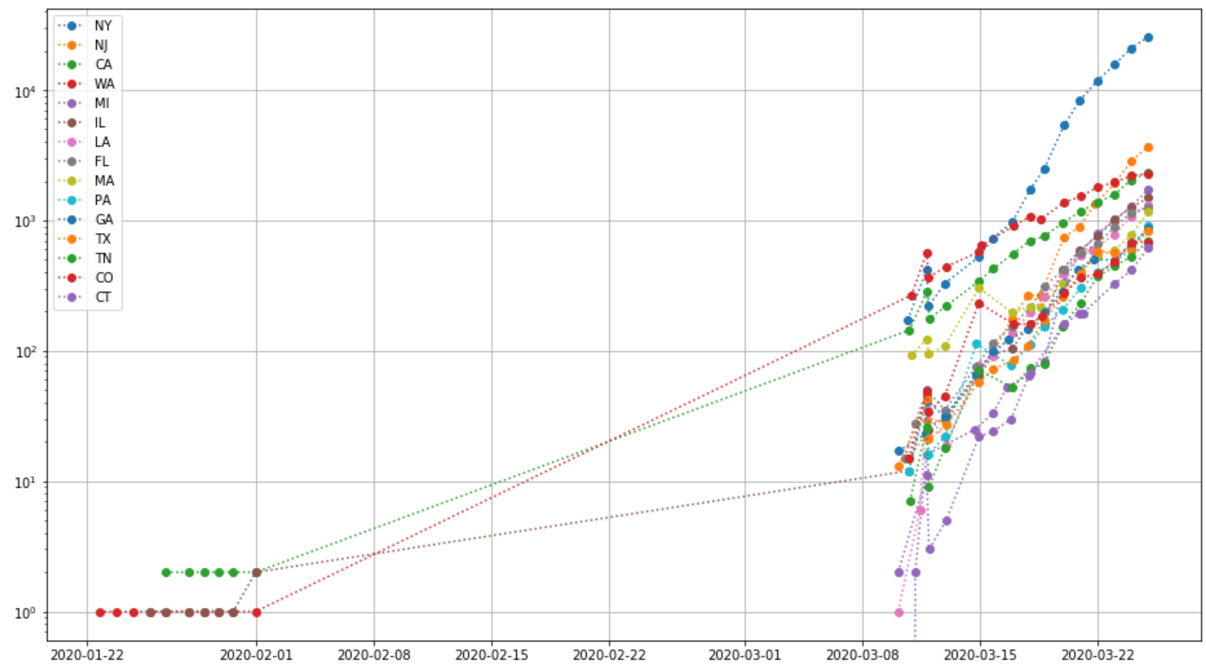
Visualization

Let's plot the `TOP_K=15` states by number of confirmed cases. **The y-axis uses a logarithmic scale in this plot.**

> To disable a logarithmic y-axis, add `logy=False` to any call to `viz_by_state()`.

In [23]:
```python
def viz_by_state(col, df, states, figsize=(16, 9), logy=False):
    from matplotlib.pyplot import figure, plot, semilogy, legend, grid
    figure(figsize=figsize)
    plotter = plot if not logy else semilogy
    for s in states:
        df0 = df[df["ST"] == s].sort_values(by="Timestamp")
        plotter(df0["Timestamp"], df0[col], "o:")
    legend(states)
    grid()

TOP_K = 15
```

In [24]:
```python
# You can modify this cell if you want to play around with the visualization.

viz_by_state("Confirmed", df_covid19_us, covid19_rankings[:TOP_K], logy=True)
```



Observe that this data is irregularly sampled and noisy. For instance, the updates do not occur every day in every state, and there are spikes due to reporting errors. Therefore, it would be useful to "smooth out" the data before plotting it, to help discern the overall trends better. That is your next task.

Filling-in missing values

We'll do a first cleaning step for you: filling-in (or *imputing*) missing daily values, so that we have at least one value per day. To see the issue more clearly, consider the data for the state of Georgia:

In [25]:
```
df_covid19_us[df_covid19_us["ST"] == "GA"].sort_values(by="Timestamp")
```

Out[25]:

	ST	Timestamp	Confirmed
133	GA	2020-03-10 03:53:03	17
134	GA	2020-03-11 18:52:03	23
135	GA	2020-03-11 20:00:00	42
136	GA	2020-03-12 21:39:10	31
137	GA	2020-03-14 17:53:03	66
138	GA	2020-03-15 18:20:19	99
139	GA	2020-03-16 16:53:06	121
140	GA	2020-03-17 20:13:22	146
141	GA	2020-03-18 19:14:34	199
142	GA	2020-03-19 21:13:35	287
143	GA	2020-03-20 19:13:30	420
144	GA	2020-03-21 17:43:03	507
145	GA	2020-03-22 23:45:00	493
146	GA	2020-03-23 23:19:34	652
147	GA	2020-03-24 23:37:31	861

There are two observations on March 11 and no observations on March 13. Suppose we want one value per day for every state. Our approach will be to *resample* the values, using pandas built-in resampler (https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling), a standard cleaning method when dealing with irregularly sampled time-series data. There are many subtle options, so we will perform one method of resampling for you. The function below implements it, storing the results in a data frame called `df_us_daily`. You do not need to understand this code right, but do run it so you can see what it will do. It will print some example results for the state of Georgia data.

In [26]:
```
def resample_daily(df):
    # This implementation is a bit weird, due to a known issue: https://github.com/pandas-dev/pandas/issues/28313
    df_r = df.sort_values(by=["ST", "Timestamp"]) \
             .set_index("Timestamp") \
             .groupby("ST", group_keys=False) \
             .resample("1D", closed="right") \
             .ffill() \
             .reset_index()
    return df_r.sort_values(by=["ST", "Timestamp"]).reset_index(drop=True)

df_us_daily = resample_daily(df_covid19_us)
df_us_daily[df_us_daily["ST"] == "GA"]
```

Out[26]:

	Timestamp	ST	Confirmed
222	2020-03-11	GA	17
223	2020-03-12	GA	42
224	2020-03-13	GA	31
225	2020-03-14	GA	31
226	2020-03-15	GA	66
227	2020-03-16	GA	99
228	2020-03-17	GA	121
229	2020-03-18	GA	146
230	2020-03-19	GA	199
231	2020-03-20	GA	287

232	2020-03-21	GA	420
233	2020-03-22	GA	507
234	2020-03-23	GA	493
235	2020-03-24	GA	652
236	2020-03-25	GA	861

Observe how there are now samples on every consecutive day beginning on March 11.

Windowed daily averages

Armed with regularly sampled data, you can now complete the next step, which is to smooth out the data using *windowed daily averages*, defined as follows.

Let c_t denote the number of confirmed cases on day t, and let d be a positive integer. Then the d-day windowed daily average on day t, denoted $\bar{c}_t$, is the mean number of confirmed cases in the d days up to and including day t. Mathematically,

$$\bar{c}_t = \frac{c_{t-(d-1)} + c_{t-(d-2)} + \cdots + c_{t-1} + c_t}{d}.$$

We'll refer to the values in the numerator as the *window* for day t.

For example, suppose $c = [0, 0, 1, 2, 2, 3, 3, 4, 6, 10]$, where the first and last values are $c_0 = 0$ and $c_9 = 10$. Now suppose $d = 3$ days. Then the windowed daily average on day t is the average of confirmed cases on days $t-2$, $t-1$, and t:

$$\bar{c}_4 = \frac{c_2 + c_3 + c_4}{3} = \frac{1+2+2}{3} = \frac{5}{3} = 1.666\ldots.$$

In this example, there aren't 3-days worth of observations for days 0 and 1. Let's treat these cases as undefined, meaning there is no average computable for those days. Therefore, the final result in this example would be

$$\bar{c} = [\text{nan}, \text{nan}, 0.333\ldots, 1.0, 1.666\ldots, 2.333\ldots, 2.666\ldots, 3.333\ldots, 4.333\ldots, 6.666\ldots],$$

where nan is a floating-point not-a-number value, which we will use a stand-in for an undefined average.

Exercise 3 (3 points): Computing windowed daily averages

Suppose you are given a data frame `df` like `df_us_daily`, which `resample_daily()` computed. That is, you may assume `df` has three columns named `"Timestamp"`, `"ST"`, and `"Confirmed"`. However, **daily observations may appear in any order within `df`.** (That is, **do not** assume they are grouped by state or sorted by timestamp *a priori*.)

Please complete the function `daily_windowed_avg(df, days)` so that it calculates the windowed daily average using windows of size `days`. Your function should return a copy of `df` with a new column named `Avg` containing this average. For days with no defined average, your function should simply omit those days from the output.

> **Note.** Although the example below shows data only for `"GA"`, the input `df` may have more than one state's worth of data in it. Therefore, your function will need to handle that case.

For example, suppose the rows in `df` with Georgia data are as follows:

Timestamp	ST	Confirmed
2020-03-12	GA	42
2020-03-17	GA	121
2020-03-11	GA	17
2020-03-15	GA	66
2020-03-18	GA	146
2020-03-16	GA	99
2020-03-13	GA	31
2020-03-14	GA	31

Observe that the rows are not necessarily in timestamp order, so you'll need to deal with that. Among these rows, the first date is March 11 and the last is March 18.

Now, suppose we use `days=3` and call your function on the full dataset (with all states), and then look at just the Georgia rows, we should see

Timestamp	ST	Confirmed	Avg
2020-03-13	GA	31	30.000000
2020-03-14	GA	31	34.6666...
2020-03-15	GA	66	42.6666...
2020-03-16	GA	99	65.3333...
2020-03-17	GA	121	95.3333...
2020-03-18	GA	146	122.0000...

You can confirm that the first day of this result, March 13, 2020, is 30, which is the average of March 11-13 (17, 42, and 31 cases, respectively). The last day, March 18, is 122, the average of March 16-18 (99, 121, and 146 cases). March 11 and 12 do not appear because they do not have three days worth of observations.

Note 0. There are many approaches to this problem. If you have good mastery of pandas, you should be able to quickly assimilate and apply its [built-in `.rolling()` technique (https://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html#window-and-resample-operations)](https://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html#window-and-resample-operations). Otherwise, it should also be straightforward to apply other techniques you already know.

Note 1. To pass the autograder, you'll need to ensure that your data frame has exactly the columns shown in the above example. (We use tibble-equivalency checks so column and row ordering does not matter.)

Note 2. The `.dtype` of columns `"Timestamp"`, `"ST"`, and `"Confirmed"` should match those of the input; the new column `"Avg"` contains floating-point values, and so should have a floating-point `.dtype`.

Note 3. Our tester already does *approximate* checking for floating-point values. Therefore, if the test code reports a mismatch, you are definitely miscalculating the averages by much more than the amount allowed by roundoff error, and you will have to keep debugging.

In [27]:
```python
def daily_windowed_avg(df, days):
    ### BEGIN SOLUTION
    return daily_windowed_avg__1(df, days)

# === Version 0: Use groupby/apply paradigm ===
def daily_window_one_df(df, days):
    from numpy import nan
    df_new = df.sort_values(by="Timestamp")
    df_new["Sums"] = df_new["Confirmed"]
    for k in range(1, days):
        df_new["Sums"].iloc[k:] += df_new["Confirmed"].iloc[:-k].values
    df_new["Sums"] /= days
    df_new.rename(columns={"Sums": "Avg"}, inplace=True)
    return df_new.iloc[days-1:]

def daily_windowed_avg__0(df, days):
    return df.groupby("ST").apply(lambda x: daily_window_one_df(x, days)).reset_index(drop=True)

# === Version 1: "Best" in terms of style ===
def daily_windowed_avg__1(df, days):
    df_avg = df.sort_values(by="Timestamp") \
               .set_index("Timestamp") \
               .groupby("ST") \
               .rolling(days) \
               .mean() \
               .reset_index() \
               .rename(columns={"Confirmed": "Avg"}) \
               .dropna()
    return df_avg.merge(df, on=["ST", "Timestamp"])

# === Version 2: Naive Loop-based ===
def daily_windowed_avg__2(df, days):
    df = df.sort_values(by=["ST", "Timestamp"])
    states_list = df["ST"].unique().tolist()
    df_new = pd.DataFrame()
    for st in states_list:
        df_st = df[df["ST"] == st]
        window = [0] * days
        for k, row in enumerate(df_st.itertuples()):
            current_day = k % days
            window[current_day] = row.Confirmed
            if k < days-1: continue
```

```
            avg = sum(window) / days
            new_row = {"ST": row.ST, "Timestamp": row.Timestamp, "Confirmed": row.Confirmed, "Avg": avg}
            df_new = df_new.append(new_row, ignore_index=True)
    df_new["Confirmed"] = df_new["Confirmed"].astype(int)
    return df_new
    ### END SOLUTION
```

In [28]:

```
# Demo of your function:
print('=== Two states: "AK" and "GA" ===')
is_ak_ga_before = df_us_daily["ST"].isin(["AK", "GA"])
display(df_us_daily[is_ak_ga_before])

print('=== Your results (days=3) ===')
df_us_daily_avg = daily_windowed_avg(df_us_daily, 3)
is_ak_ga_after = df_us_daily_avg["ST"].isin(["AK", "GA"])
display(df_us_daily_avg[is_ak_ga_after])
```

```
=== Two states: "AK" and "GA" ===
```

	Timestamp	ST	Confirmed
0	2020-03-11	AK	0
1	2020-03-12	AK	1
2	2020-03-13	AK	1
3	2020-03-14	AK	1
4	2020-03-15	AK	1
5	2020-03-16	AK	1
6	2020-03-17	AK	1
7	2020-03-18	AK	3
8	2020-03-19	AK	6
9	2020-03-20	AK	9
10	2020-03-21	AK	12
11	2020-03-22	AK	15
12	2020-03-23	AK	19
13	2020-03-24	AK	30
14	2020-03-25	AK	34
222	2020-03-11	GA	17
223	2020-03-12	GA	42
224	2020-03-13	GA	31
225	2020-03-14	GA	31
226	2020-03-15	GA	66
227	2020-03-16	GA	99
228	2020-03-17	GA	121
229	2020-03-18	GA	146
230	2020-03-19	GA	199
231	2020-03-20	GA	287
232	2020-03-21	GA	420
233	2020-03-22	GA	507
234	2020-03-23	GA	493
235	2020-03-24	GA	652
236	2020-03-25	GA	861

```
=== Your results (days=3) ===
```

	ST	Timestamp	Avg	Confirmed
0	AK	2020-03-13	0.666667	1
1	AK	2020-03-14	1.000000	1

2	AK	2020-03-15	1.000000	1
3	AK	2020-03-16	1.000000	1
4	AK	2020-03-17	1.000000	1
5	AK	2020-03-18	1.666667	3
6	AK	2020-03-19	3.333333	6
7	AK	2020-03-20	6.000000	9
8	AK	2020-03-21	9.000000	12
9	AK	2020-03-22	12.000000	15
10	AK	2020-03-23	15.333333	19
11	AK	2020-03-24	21.333333	30
12	AK	2020-03-25	27.666667	34
204	GA	2020-03-13	30.000000	31
205	GA	2020-03-14	34.666667	31
206	GA	2020-03-15	42.666667	66
207	GA	2020-03-16	65.333333	99
208	GA	2020-03-17	95.333333	121
209	GA	2020-03-18	122.000000	146
210	GA	2020-03-19	155.333333	199
211	GA	2020-03-20	210.666667	287
212	GA	2020-03-21	302.000000	420
213	GA	2020-03-22	404.666667	507
214	GA	2020-03-23	473.333333	493
215	GA	2020-03-24	550.666667	652
216	GA	2020-03-25	668.666667	861

In [29]:

```
# Test cell: `ex3__daily_windowed_avg` (3 points)

### BEGIN HIDDEN TESTS
def ex3_gen_soln(force=False):
    def ex3_soln(df, days):
        df_avg = df.sort_values(by="Timestamp") \
                   .set_index("Timestamp") \
                   .groupby("ST") \
                   .rolling(days) \
                   .mean() \
                   .reset_index() \
                   .rename(columns={"Confirmed": "Avg"}) \
                   .dropna()
        return df_avg.merge(df, on=["ST", "Timestamp"])
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    #
    from os.path import isfile
    soln_file = get_path('covid19/ex3_soln.csv')
    if isfile(soln_file) and not force:
        print(f"Solution file, '{soln_file}', already exists. NOT regenerating...")
    else:
```

```
        print(f"Generating solution file, '{soln_file}'...")
        df = ex3_soln(df_us_daily, 3)
        df.to_csv(soln_file, index=False)
        print("==> Done!")

ex3_gen_soln(force=False)
### END HIDDEN TESTS

def ex3_gen_state_df(st, days):
    from random import randint, random
    from pandas import DataFrame, concat
    from problem_utils import ex0_random_date
    def rand_day():
        from datetime import datetime
        date = ex0_random_date()
        return datetime(date.year, date.month, date.day)
    def inc_date(date, days=1):
        from datetime import timedelta
        return date + timedelta(days=days)
    dates = []
    sts = []
    confs = [randint(1, 10)]
    avgs = []
    r0 = 1 + random()
    day = rand_day()
    num_days = days + randint(1, 10)
    for k in range(num_days):
        dates.append(day)
        sts.append(st)
        confs.append(int(confs[0] * (r0**k)))
        if k >= days-1: avgs.append(sum(confs[(-days):]) / days)
        day = inc_date(day)
    df = DataFrame({"Timestamp": dates,
                    "ST": sts,
                    "Confirmed": confs[1:]})
    df_soln = DataFrame({"Timestamp": dates[(days-1):],
                         "ST": sts[(days-1):],
                         "Confirmed": confs[days:],
                         "Avg": avgs})
    return df, df_soln

def ex3_gen_df():
    from random import randint, sample
    from pandas import concat
    num_states = randint(1, 4)
    days = randint(1, 4)
    states = sample(STATE_NAMES["Abbrv"].tolist(), num_states)
    df_list = []
    df_soln_list = []
    for st in states:
        df_st, df_st_soln = ex3_gen_state_df(st, days)
        df_list.append(df_st)
        df_soln_list.append(df_st_soln)
    df = concat(df_list, ignore_index=True).sample(frac=1).reset_index(drop=True)
    df_soln = concat(df_soln_list, ignore_index=True).sort_values(by=["ST", "Timestamp"])
    try:
        df_your_soln = daily_windowed_avg(df, days)
        assert_tibbles_are_equivalent(df_soln, df_your_soln)
    except:
        print("\n*** ERROR ***")
        print("Input data frame:")
        display(df)
        print(f"Expected solution (days={days}):")
        display(df_soln)
        print("Your solution:")
        display(df_your_soln)
        raise

for trial in range(10):
    print(f"=== Trial #{trial} / 9 ===")
    ex3_gen_df()

print("\n(Passed.)")
```

```
Solution file, './resource/asnlib/publicdata/covid19/ex3_soln.csv', already exists. NOT regenerating...
=== Trial #0 / 9 ===
=== Trial #1 / 9 ===
=== Trial #2 / 9 ===
=== Trial #3 / 9 ===
=== Trial #4 / 9 ===
=== Trial #5 / 9 ===
```

```
=== Trial #6 / 9 ===
=== Trial #7 / 9 ===
=== Trial #8 / 9 ===
=== Trial #9 / 9 ===

(Passed.)
```

(In case Exercise 3 isn't working) Daily windowed averages

In case you can't get a working solution to Exercise 3, we have pre-computed the daily windowed averages. The code cell below reads this data and stores it in the variable, `df_us_daily_avg`. You will need it in the subsequent exercises, so do not modify it!

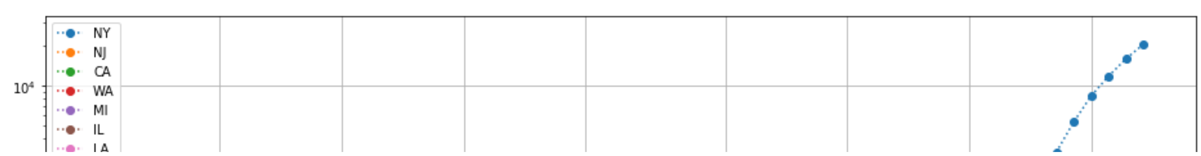
In [30]:
```python
with open(get_path('covid19/ex3_soln.csv'), "rt") as fp:
    df_us_daily_avg = pd.read_csv(get_path('covid19/ex3_soln.csv'), parse_dates=["Timestamp"])
df_us_daily_avg[df_us_daily_avg["ST"].isin(["AK", "GA"])]
```

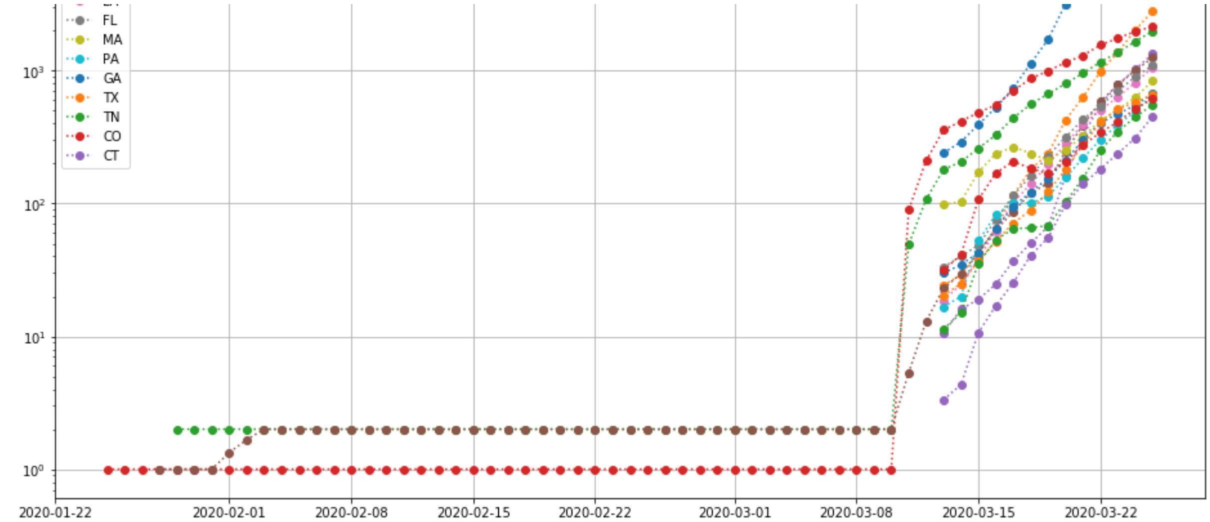
Out[30]:

	ST	Timestamp	Avg	Confirmed
0	AK	2020-03-13	0.666667	1
1	AK	2020-03-14	1.000000	1
2	AK	2020-03-15	1.000000	1
3	AK	2020-03-16	1.000000	1
4	AK	2020-03-17	1.000000	1
5	AK	2020-03-18	1.666667	3
6	AK	2020-03-19	3.333333	6
7	AK	2020-03-20	6.000000	9
8	AK	2020-03-21	9.000000	12
9	AK	2020-03-22	12.000000	15
10	AK	2020-03-23	15.333333	19
11	AK	2020-03-24	21.333333	30
12	AK	2020-03-25	27.666667	34
204	GA	2020-03-13	30.000000	31
205	GA	2020-03-14	34.666667	31
206	GA	2020-03-15	42.666667	66
207	GA	2020-03-16	65.333333	99
208	GA	2020-03-17	95.333333	121
209	GA	2020-03-18	122.000000	146
210	GA	2020-03-19	155.333333	199
211	GA	2020-03-20	210.666667	287
212	GA	2020-03-21	302.000000	420
213	GA	2020-03-22	404.666667	507
214	GA	2020-03-23	473.333333	493
215	GA	2020-03-24	550.666667	652
216	GA	2020-03-25	668.666667	861

Here is a visualization of the daily averages, which should appear smoother. As such, the trends should be a little more clear as well.

In [31]:
```python
# You can modify this cell if you want to play around with the visualization.

viz_by_state("Avg", df_us_daily_avg, covid19_rankings[:TOP_K], logy=True)
```





Step 2: Flights (re-)analysis

Recall from Notebook 11 that you used a Markov chain-based model to "rank" airport networks by how likely a certain "random flyer" is to end up at each airport. In this final step of this problem, you'll apply a similiar idea to rank states, and see how well it correlates with the state-by-state numbers of confirmed COVID-19 cases.

Raw data. First, observe that our raw data differs slightly from Notebook 11. It consists of all flights from calendar year 2019 (the latest available from the original source, as no 2020 flights are present there), and we've added a column with each airport's two-letter state postal code. Let's load these flights into a DataFrame called `flights`. (You don't need to understand this code in depth, but do pay attention to the format of the output sample from `flights`.)

In [32]:
```
def load_flights(infile=get_path('us-flights/us-flights-2019--86633396_T_ONTIME_REPORTING.csv')):
    keep_cols = ["FL_DATE", "ORIGIN_STATE_ABR", "DEST_STATE_ABR", "OP_UNIQUE_CARRIER", "OP_CARRIER_FL_NUM"]
    flights = pd.read_csv(infile)[keep_cols]
    us_sts = set(STATE_NAMES["Abbrv"])
    origin_is_state = flights['ORIGIN_STATE_ABR'].isin(us_sts)
    dest_is_state = flights['DEST_STATE_ABR'].isin(us_sts)
    return flights.loc[origin_is_state & dest_is_state].copy()

flights = load_flights()
print(f"There are {len(flights):,} direct flight segments in the `flights` data frame.")
print("Here are the first few:")
flights.head()
```

```
There are 7,352,434 direct flight segments in the `flights` data frame.
Here are the first few:
```

Out[32]:

	FL_DATE	ORIGIN_STATE_ABR	DEST_STATE_ABR	OP_UNIQUE_CARRIER	OP_CARRIER_FL_NUM
0	2019-01-01	FL	NY	9E	5122
1	2019-01-01	NY	VA	9E	5123
2	2019-01-01	NC	GA	9E	5130
3	2019-01-01	GA	AL	9E	5136
4	2019-01-01	AL	GA	9E	5136

Outdegrees. In Notebook 11, we calculated the outdegree of each airport u to be the number of distinct endpoints (other airports) reachable from u.

For the analysis in this problem, we will use a *different* definition for the outdegree. In particular, we'll define the outdegree d_u of **state** u (e.g., the state of Georgia, the state of California) to be the total number of direct flight segments from state u to all other states. In pandas, we can use a group-by-count aggregation to compute these outdegrees. Here is some code that does so, producing a data frame named `outdegrees` with two columns, the origin state (`"Origin"`) and outdegree value (`"Outdegree"`), sorted in descending order of outdegree. (You should be able to understand this code, which may help you in the next exercise.)

In [33]:
```
def calc_outdegrees(flights):
    outdegrees = flights[['ORIGIN_STATE_ABR', 'DEST_STATE_ABR']] \
                 .groupby(['ORIGIN_STATE_ABR']) \
                 .count() \
```

```
                    .count() \
                    .reset_index() \
                    .rename(columns={'ORIGIN_STATE_ABR': 'Origin',
                                     'DEST_STATE_ABR': 'Outdegree'}) \
                    .sort_values(by='Outdegree', ascending=False) \
                    .reset_index(drop=True)
    return outdegrees

# Demo:
outdegrees = calc_outdegrees(flights)
print(f"There are {len(outdegrees)} states with a non-zero outdegree.")
print("Here are the first ten:")
outdegrees.head(10)
```

```
There are 49 states with a non-zero outdegree.
Here are the first ten:
```

Out[33]:

	Origin	Outdegree
0	CA	814858
1	TX	802044
2	FL	569062
3	IL	442091
4	GA	419791
5	NY	385272
6	NC	339959
7	CO	282450
8	VA	270206
9	AZ	204998

Exercise 4 (2 points): State transition probabilities

To run the ranking analysis, recall that we need to construct a probability transition matrix. For our state-to-state analysis, we therefore wish to estimate the probability of going from state i to state j. Let's define that probability to be the number of direct flight segments from state i to state j divided by the outdegree of state i.

Complete the function, `calc_state_trans_probs(flights, outdegrees)` to compute these state-to-state transition probabilities. Your function should accept two data frames like `flights` and `outdegrees` as defined above. In particular, you may assume the following;

- The `flights` data frame has three columns: "`ORIGIN_STATE_ABR`" (originating state, a two-letter abbreviation), "`DEST_STATE_ABR`" (destination state abbreviation), and "`FL_DATE`" (date of direct flight).
- The `outdegrees` data frame has two columns: "`Origin`" (originating state, a two-letter abbreviation) and "`Outdegree`" (an integer).

Your function should and return a new data frame with exactly these columns:

- "`Origin`": The origin state, i.e., state i, as a two-letter abbreviation.
- "`Dest`": The destination state, i.e., state j, as a two-letter abbreviation.
- "`Count`": The number of direct flight segments from state i to state j.
- "`TransProb`": The transition probability of going from state i to state j, i.e., the count divided by the outdegree.

In [34]:
```
def calc_state_trans_probs(flights, outdegrees):
    ### BEGIN SOLUTION
    probs = flights[['ORIGIN_STATE_ABR', 'DEST_STATE_ABR', 'FL_DATE']] \
            .groupby(['ORIGIN_STATE_ABR', 'DEST_STATE_ABR']) \
            .count() \
            .reset_index() \
            .rename(columns={'ORIGIN_STATE_ABR': 'Origin',
                             'DEST_STATE_ABR': 'Dest',
                             'FL_DATE': 'Count'}) \
            .merge(outdegrees, on='Origin', how='inner')
    probs['TransProb'] = probs['Count'] / probs['Outdegree']
    del probs['Outdegree']
    return probs
    ### END SOLUTION
```

In [35]:
```
# Demo, Part 0:
probs = calc_state_trans_probs(flights, outdegrees)
print(f"There are {len(probs)} state-to-state transition probabilities in your result.")
print("Here are ten with the largest transition probabilities:")
```

```
display(probs.sort_values(by="TransProb", ascending=False).head(10))
```

```
There are 1293 state-to-state transition probabilities in your result.
Here are ten with the largest transition probabilities:
```

	Origin	Dest	Count	TransProb
287	HI	HI	68017	0.593315
0	AK	AK	22914	0.575714
722	ND	MN	8708	0.508022
939	OR	CA	33501	0.440252
658	MS	TX	6673	0.426717
933	OK	TX	18488	0.425746
1283	WY	CO	3714	0.425040
42	AR	TX	13388	0.420927
810	NM	TX	10355	0.374340
457	LA	TX	31051	0.359511

In [36]:

```
# Demo, Part 1:

print("""
As a sanity check, let's see if the sum of all outgoing links per state
is (approximately) 1.0. If it isn't, meaning any of the rows of the
output below are `False`, use that information to help yourself debug.
""")
sanity = (probs[['Origin', 'TransProb']].groupby('Origin').sum() - 1.0).abs() < 1e-14
sanity
```

```
As a sanity check, let's see if the sum of all outgoing links per state
is (approximately) 1.0. If it isn't, meaning any of the rows of the
output below are `False`, use that information to help yourself debug.
```

Out[36]:

	TransProb
Origin	
AK	True
AL	True
AR	True
AZ	True
CA	True
CO	True
CT	True
FL	True
GA	True
HI	True
IA	True
ID	True
IL	True
IN	True
KS	True
KY	True
LA	True
MA	True
MD	True
ME	True
MI	True

MN	True
MO	True
MS	True
MT	True
NC	True
ND	True
NE	True
NH	True
NJ	True
NM	True
NV	True
NY	True
OH	True
OK	True
OR	True
PA	True
RI	True
SC	True
SD	True
TN	True
TX	True
UT	True
VA	True
VT	True
WA	True
WI	True
WV	True
WY	True

In [37]:

```
# Test cell: `ex4__calc_state_trans_probs` (2 points)

def ex4_gen_df_st(st):
    from random import randint, random, choice
    from collections import defaultdict
    from problem_utils import ex0_random_date
    from pandas import DataFrame
    states = list(STATE_NAMES["Abbrv"])
    num_unique_edges = randint(1, 4)
    dates = []
    dests = []
    counts = defaultdict(int)
    outdegree = 0
    for _ in range(num_unique_edges):
        if random() < 0.33:
            num_reps = randint(2, 4)
        else:
            num_reps = 1
        dest_st = choice(states)
        dests += [dest_st] * num_reps
        dates += [ex0_random_date() for _ in range(num_reps)]
        counts[(st, dest_st)] += num_reps
        outdegree += num_reps
    flights = DataFrame({"FL_DATE": dates,
                         "ORIGIN_STATE_ABR": [st] * len(dates),
                         "DEST_STATE_ABR": dests})
    dests_st = []
    counts_st = []
    probs_st = []
    for (st, dest_st), c in counts.items():
        dests_st.append(dest_st)
```

```
        counts_st.append(c)
        probs_st.append(c / outdegree)
    sts = [st] * len(dests_st)
    probs = DataFrame({"Origin": sts,
                       "Dest": dests_st,
                       "Count": counts_st,
                       "TransProb": probs_st})
    return flights, probs, outdegree

def ex4_check_one():
    from random import randint, sample
    from pandas import DataFrame, concat
    num_states = randint(1, 4)
    states = list(STATE_NAMES["Abbrv"])
    flights_list = []
    probs_list = []
    outdegrees_list = []
    sts = sample(states, num_states)
    for st in sts:
        flights_st, probs_st, outdegree_st = ex4_gen_df_st(st)
        flights_list.append(flights_st)
        probs_list.append(probs_st)
        outdegrees_list.append(outdegree_st)
    flights = concat(flights_list, ignore_index=True) \
              .sort_values(by="FL_DATE") \
              .reset_index(drop=True)
    probs = concat(probs_list, ignore_index=True) \
            .sort_values(by="Origin") \
            .reset_index(drop=True)
    outdegrees = DataFrame({"Origin": sts,
                            "Outdegree": outdegrees_list})
    try:
        your_probs = calc_state_trans_probs(flights, outdegrees)
        assert_tibbles_are_equivalent(probs, your_probs)
    except:
        print("\n*** ERROR ***\n")
        print("`flights` input:")
        display(flights)
        print("`outdegrees` input:")
        display(outdegrees)
        print("Expected output:")
        display(probs)
        print("Your output:")
        display(your_probs)
        raise

for trial in range(10):
    print(f"=== Trial #{trial} / 9 ===")
    ex4_check_one()

EXERCISE4_PASSED = True
print("\n(Passed.)")
```

```
=== Trial #0 / 9 ===
=== Trial #1 / 9 ===
=== Trial #2 / 9 ===
=== Trial #3 / 9 ===
=== Trial #4 / 9 ===
=== Trial #5 / 9 ===
=== Trial #6 / 9 ===
=== Trial #7 / 9 ===
=== Trial #8 / 9 ===
=== Trial #9 / 9 ===

(Passed.)
```

(In case Exercise 4 isn't working)

The rest of this notebook completes the comparison between state-rankings by confirmed cases and those by the airport network. It does depend on a working Exercise 4. However, running it is for your edification only, as there are no additional exercises or test cells below. Nevertheless, if the autograder has trouble completing due to errors in the code below, you can try converting the code cells to Markdown (effectively disabling them) and see if that helps.

State rankings. The next code cell runs the PageRank-style algorithm on the state-to-state airport network and produces a ranking. It depends on a correct result for Exercise 4, so if yours is not working completely, it might not run to completion. If that causes issues with the autograder, you can try converting the cell to Markdown to (effectively) disable it.

In [38]:
```
def spy(A, figsize=(6, 6), markersize=0.5):
    """Visualizes a sparse matrix."""
    from matplotlib.pyplot import figure, spy, show
    fig = figure(figsize=figsize)
    spy(A, markersize=markersize)
    show()

def display_vec_sparsely(x, name='x'):
    from numpy import argwhere
    from pandas import DataFrame
    i_nz = argwhere(x).flatten()
    df_x_nz = DataFrame({'i': i_nz, '{}[i] (non-zero only)'.format(name): x[i_nz]})
    display(df_x_nz.head(5))
    if len(df_x_nz) > 5:
        print("...")
        display(df_x_nz.tail(5))

def eval_markov_chain(P, x0, t_max):
    x = x0
    for t in range(t_max):
        x = P.T.dot(x)
    return x

def rank_states_by_air_network(probs, t_max=100, verbose=True):
    from numpy import array, zeros, ones, argsort, arange
    from scipy.sparse import coo_matrix
    from pandas import DataFrame

    # Create transition matrix
    unique_origins = set(probs['Origin'])
    unique_dests = set(probs['Dest'])
    unique_states = array(sorted(unique_origins | unique_dests))
    state_ids = {st: i for i, st in enumerate(unique_states)}
    num_states = max(state_ids.values()) + 1

    s2s = probs.copy()
    s2s['OriginID'] = s2s['Origin'].map(state_ids)
    s2s['DestID'] = s2s['Dest'].map(state_ids)

    P = coo_matrix((s2s['TransProb'], (s2s['OriginID'], s2s['DestID'])),
                   shape=(num_states, num_states))
    if verbose: spy(P)

    # Run ranking algorithm
    x0 = zeros(num_states)
    x0[state_ids['WA']] = 1.0 # First state to report confirmed COVID-19 cases

    if verbose:
        print("Initial condition:")
        display_vec_sparsely(x0, name='x0')

    x = eval_markov_chain(P, x0, t_max)

    if verbose:
        print("Final probabilities:")
        display_vec_sparsely(x)

    # Produce a results table of rank-ordered states
    ranks = argsort(-x)
    df_ranks = DataFrame({'Rank': arange(1, len(ranks)+1),
                          'State': unique_states[ranks],
                          'x(t)': x[ranks]})
    df_ranks['ID'] = df_ranks['State'].map(state_ids)

    return df_ranks

if "EXERCISE4_PASSED" in dir() and EXERCISE4_PASSED:
    print("Running the ranking algorithm...")
    airnet_rankings = rank_states_by_air_network(probs, verbose=False)

    print(f"==> Here are the top-{TOP_K} states:")
    display(airnet_rankings.head(TOP_K))
else:
    print("We did not detect that the Exercise 4 test cell passed, so we aren't running this cell.")
```

```
Running the ranking algorithm...
==> Here are the top-15 states:
```

	Rank	State	x(t)	ID

0	1	CA	0.110829	4
1	2	TX	0.109096	41
2	3	FL	0.077404	7
3	4	IL	0.060123	12
4	5	GA	0.057099	8
5	6	NY	0.052398	32
6	7	NC	0.046237	25
7	8	CO	0.038418	5
8	9	VA	0.036742	43
9	10	AZ	0.027885	3
10	11	MI	0.027664	20
11	12	PA	0.025440	36
12	13	NV	0.025127	31
13	14	MN	0.023198	21
14	15	WA	0.021942	45

Comparing the two rankings. We now have a ranking of states by number of confirmed COVID-19 cases, as well as a separate ranking of states by air-network connectivity. To compare them, we'll use a measure called *rank-biased overlap* (RBO) (https://doi.org/10.1145/1852102.1852106). Very roughly speaking, this measure is an estimate of the probability that a reader comparing the top few entries of two rankings tends to encounter the same items, so a value closer to 1 means the top entries of the two rankings are more similar.

> **Note 0.** We say "top few" above because RBO is parameterized by a "patience" parameter, which is related to how many of the top entries the reader will inspect before stopping. The reason for this parameter originates in the motivation for RBO, which was to measure the similarity between search engine results. The code we are using to calculate RBO uses this implementation (https://github.com/dlukes/rbo)).
>
> **Note 1.** This cell should only be run if Exercise 4 passes.

In [39]:
```
from rbo import rbo

if "EXERCISE4_PASSED" in dir() and EXERCISE4_PASSED:
    compare_rankings = rbo(covid19_rankings, # ranking by confirmed COVID-19 cases
                           airnet_rankings['State'].values, # ranking by air-network connectivity
                           0.95) # "patience" parameter
    print(f"Raw RBO result: {compare_rankings}\n\n==> RBO score is {compare_rankings.ext:.3}")
else:
    print("We did not detect that the Exercise 4 test cell passed, so we aren't running this cell.")
```

```
Raw RBO result: RBO(min=0.5783307042800485, res=0.02058258852563008, ext=0.5987896438055367)

==> RBO score is 0.599
```

If everything is correct, you'll see an RBO score of around 0.6, which suggests that the connectivity of the airport network may help explain the number of confirmed COVID-19 cases we are seeing in each state.

Fin! You've reached the end of this problem. Don't forget to restart and run all cells again to make sure it's all working when run in sequence; and make sure your work passes the submission process. Good luck!

Problem 1: Shortest paths (total value: 6 points)

Version 1.0

As a data analyst in the field, you will spend the bulk of your time just trying to get data into a form that is useful for analysis. This problem assess your ability to do that.

In particular, you will calculate shortest paths in the California road network. You will use SQL, pandas, and basic Python to transform the data so that it can be fed into NetworkX (https://networkx.github.io/), a Python module for analyzing and visualizing graphs or networks.

This notebook has four (4) exercises, numbered 0 through 3, worth a total of **6 points**. Exercises 0 and 3 are worth 2 points each, and Exercises 1 and 2 are worth 1 point each.

All exercises are independent, so if you get stuck on one, try moving on to the next one. However, in such cases do look for notes labeled, *"In case Exercise XXX isn't working"*, as you may need to run some code cells that load pre-computed results that will allow you to continue with any subsequent exercises.

Pro-tips.

- If your program behavior seem strange, try resetting the kernel and rerunning everything.
- If you mess up this notebook or just want to start from scratch, save copies of all your partial responses and use `Actions` → `Reset Assignment` to get a fresh, original copy of this notebook. (*Resetting will wipe out any answers you've written so far, so be sure to stash those somewhere safe if you intend to keep or reuse them!*)
- If you generate excessive output (e.g., from an ill-placed `print` statement) that causes the notebook to load slowly or not at all, use `Actions` → `Clear Notebook Output` to get a clean copy. The clean copy will retain your code but remove any generated output. **However**, it will also **rename** the notebook to `clean.xxx.ipynb`. Since the autograder expects a notebook file with the original name, you'll need to rename the clean notebook accordingly.

Good luck!

Background: The shortest path problem

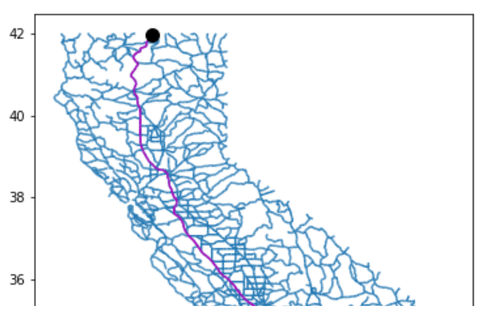
The dataset in this problem is the California road network. It consists of **intersections**, which are point locations on the map having (x, y) coordinates, and **road (segments)**, each of which connects a pair of intersections. In the language of graphs or networks, we refer to intersections as *nodes* and road segments as *edges*.

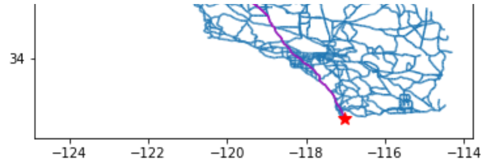
Run this code cell to see a picture of your task:

In [1]:
```
from problem_utils import get_path, display_image, assert_tibbles_are_equivalent, pandas_df_to_markdown_table
print("Example of what you will produce in this problem (shortest paths on the California road network):")
display_image(get_path('ca-roads-path-demo.png'))
```

```
Example of what you will produce in this problem (shortest paths on the California road network):
```

Out[1]:





The blue lines are road segments. The two markers (black solid circle and red star) are a source and destination pair. The purple route connecting this pair is a path, which is a list of connected road segments. The one you see in the image happens to be the "shortest path" between the source and destination.

Your task in this notebook. NetworkX can calculate the shortest path, but you have to supply the network. The exercises below walk you through the process of extracting the network data from a SQL database and transforming it so that NetworkX can use it.

Preliminaries. Run the following code cell, which will pre-load some modules you'll need for this problem.

In [2]:
```
import sys

import sqlite3 as db
import pandas as pd

%matplotlib inline
import matplotlib.pyplot as plt

print(f"* Python version:\n{sys.version}\n")
print(f"* sqlite3 version: {db.version}")
print(f"* pandas version: {pd.__version__}")
```

```
* Python version:
3.7.5 (default, Dec 18 2019, 06:24:58)
[GCC 5.5.0 20171010]

* sqlite3 version: 2.6.0
* pandas version: 0.25.3
```

Opening the database

Let's start by opening a read-only connection to the road network database:

In [3]:
```
conn = db.connect('file:' + get_path('ca-roads/network.db') + '?mode=ro', uri=True)
```

This database has two tables: `Intersections` and `Roads`. Let's take a look.

Intersections. The `Intersections` table looks like this:

In [4]:
```
pd.read_sql_query("SELECT * FROM Intersections LIMIT 7", conn)
```

Out[4]:

	ID	X	Y
0	0	-121.904167	41.974556
1	1	-121.902153	41.974766
2	2	-121.896790	41.988075
3	3	-121.889603	41.998032
4	4	-121.886681	42.008739
5	5	-121.915062	41.970314
6	6	-121.910088	41.973942

Each intersection is a row in the database. It has a unique integer identifier (the `"ID"` column) and (x, y)-coordinates (the `"X"` and `"Y"` columns). You'll need the latter to visualize the paths.

Roads. The `Roads` table looks like this:

In [5]:
```
pd.read_sql_query("SELECT * FROM Roads LIMIT 5", conn)
```

Out[5]:

	ID	AID	BID
0	0	0	1
1	1	0	6
2	2	1	2
3	3	2	3
4	4	3	4

Each road (segment) is a row of the table, with a unique integer ID (the `"ID"` column). It **connects** two intersections, and you can think of them as "point A" and "point B." The IDs of these two points are `"AID"` and `"BID"`.

> For example, row 1 of `Roads` is a road segment whose ID is also 1. It connects intersections 0 and 6. Recalling the `Intersections` table from before, intersection 0 has physical coordinates of (-121.904167, 41.974556), and intersection 6 has coordinates (-121.910088, 41.973942).

Exercise 0 (2 points): Querying the roads geometry

Let's pull the road and intersection data together. In the code cell below, create a query string named `query_roads` that will produce an output table with exactly the following 7 columns:

- `E`: The road segment ID
- `A`: Point A of the road segment
- `AX`: The x-coordinate of point A
- `AY`: The y-coordinate of point A
- `B`: Point B of the road segment
- `BX`: The x-coordinate of point B
- `BY`: The y-coordinate of point B

The "demo" below will run your query string against the database, returning the result in a pandas `DataFrame` object named `df_roads`.

For example, a few rows of your output should be:

E	A	AX	AY	B	BX	BY
0	0	-121.90416699999999	41.974556	1	-121.902153	41.974765999999995
1	0	-121.90416699999999	41.974556	6	-121.91008799999999	41.973942
2	1	-121.902153	41.974765999999995	2	-121.89679	41.988075
3	2	-121.89679	41.988075	3	-121.88960300000001	41.998032
4	3	-121.88960300000001	41.998032	4	-121.88668100000001	42.008739

...

In [6]:
```
# Define a variable named `query_roads`:
### BEGIN SOLUTION
query_roads = """
SELECT R.ID AS E,
       IA.ID AS A, IA.X AS AX, IA.Y AS AY,
       IB.ID AS B, IB.X AS BX, IB.Y AS BY
    FROM Roads AS R, Intersections AS IA, Intersections AS IB
    WHERE A=AID AND B=BID
"""
### END SOLUTION

# Demo of your query:
df_roads = pd.read_sql_query(query_roads, conn)
df_roads.head()
```

Out[6]:

	E	A	AX	AY	B	BX	BY
0	0	0	-121.904167	41.974556	1	-121.902153	41.974766
1	1	0	-121.904167	41.974556	6	-121.910088	41.973942
2	2	1	-121.902153	41.974766	2	-121.896790	41.988075

3	3	2	-121.896790	41.988075	3	-121.889603	41.998032
4	4	3	-121.889603	41.998032	4	-121.886681	42.008739

In [7]:
```
# Test cell: `ex0__query_roads` (2 points)

### BEGIN HIDDEN TESTS
def ex0_soln(conn):
    from pandas import read_sql_query
    df = read_sql_query("SELECT R.ID AS E, IA.ID AS A, IA.X AS AX, IA.Y AS AY, IB.ID AS B, IB.X AS BX, IB.Y AS BY F
ROM Roads AS R, Intersections AS IA, Intersections AS IB WHERE A=AID AND B=BID", conn)
    return df

#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#

def ex0_gen_soln(force=False):
    from problem_utils import get_path
    from os.path import isfile
    from problem_utils import pandas_df_to_markdown_table

    soln_file = get_path('ca-roads/ex0_soln.csv')
    if force or not isfile(soln_file):
        print(f"Generating solution file, '{soln_file}'...")
        ex0_soln(conn).to_csv(soln_file, index=False)
        print("=== Demo ===")
        print(pandas_df_to_markdown_table(df_roads.head(), index=False))
    else:
        print(f"Solution file, '{soln_file}', already exists; skipping re-generation...")

ex0_gen_soln()
### END HIDDEN TESTS

def ex0_gen_dfs():
    global query_roads # Your query
    from random import randint, random, sample
    from itertools import combinations
    from pandas import DataFrame

    num_ints = randint(3, 5)
    ids = list(range(num_ints))
    xs, ys = [], []
    for v in ids:
        xs.append(-1 + 2*random())
        ys.append(-1 + 2*random())
    df_ints = DataFrame({"ID": ids, "X": xs, "Y": ys})

    all_roads = list(combinations(ids, 2))
    num_roads = randint(1, len(all_roads))
    roads = set()
    for a, b in sample(all_roads, num_roads):
        roads |= {(a, b)}
    rr, aa, bb = [], [], []
    for r, (a, b) in enumerate(roads):
        rr += [r]
        aa += [a]
        bb += [b]
    axs = [xs[a] for a in aa]
    ays = [ys[a] for a in aa]
    bxs = [xs[b] for b in bb]
    bys = [ys[b] for b in bb]
```

```
    df_roads = DataFrame({"ID": rr, "AID": aa, "BID": bb})
    df_soln = DataFrame({"E": rr,
                         "A": aa, "AX": axs, "AY": ays,
                         "B": bb, "BX": bxs, "BY": bys})
    return df_ints, df_roads, df_soln

def ex0_check_one():
    from sqlite3 import connect
    from pandas import read_sql_query

    # Randomly generate a sample problem (and solution)
    df_ints, df_roads, df_soln = ex0_gen_dfs()

    # Create a database
    db_conn = connect(":memory:")
    df_ints.to_sql("Intersections", db_conn)
    df_roads.to_sql("Roads", db_conn)

    # Try your query
    try:
        df_your_soln = read_sql_query(query_roads, db_conn)
        assert_tibbles_are_equivalent(df_soln, df_your_soln)
    except:
        print("\n*** ERROR ***\n")
        print("Your code did not produce the expected result on a randomly generated input.")
        print("==> Intersections:")
        display(df_ints)
        print("==> Roads:")
        display(df_roads)
        print("==> Expected solution:")
        display(df_soln)
        print("==> Your solution:")
        display(df_your_soln)
        raise
    finally:
        db_conn.close()

print()
for trial in range(10):
    print(f"=== Trial #{trial} / 9 ===")
    ex0_check_one()

print("\n(Passed.)")
```

```
Solution file, './resource/asnlib/publicdata/ca-roads/ex0_soln.csv', already exists; skipping re-generation...

=== Trial #0 / 9 ===
=== Trial #1 / 9 ===
=== Trial #2 / 9 ===
=== Trial #3 / 9 ===
=== Trial #4 / 9 ===
=== Trial #5 / 9 ===
=== Trial #6 / 9 ===
=== Trial #7 / 9 ===
=== Trial #8 / 9 ===
=== Trial #9 / 9 ===

(Passed.)
```

A pre-generated road network

Whether you solved Exercise 0 or not, the following code cell will load a pre-generated solution for these data and store them in a pandas `DataFrame` called `df_roads`, so you can continue with the problem. Run this cell now. Subsequent exercises depend on `df_roads`, so do **not** modify it.

In [8]:
```
df_roads = pd.read_csv(get_path("ca-roads/ex0_soln.csv"))
display(df_roads.head())
print("...")
```

	E	A	AX	AY	B	BX	BY
0	0	0	-121.904167	41.974556	1	-121.902153	41.974766
1	1	0	-121.904167	41.974556	6	-121.910088	41.973942
2	2	1	-121.902153	41.974766	2	-121.896790	41.988075
3	3	2	-121.896790	41.988075	3	-121.889603	41.998032

4	4	3	-121.889603	41.998032	4	-121.886681	42.008739

...

Basic visualization

Before the next exercise, run this cell to define some code that will help us draw the road network.

In [9]:
```
def node_coords(sx, sy):
    return [(x, y) for x, y in zip(sx, sy)]

def edge_coords(axy, bxy):
    return [[a, b] for a, b in zip(axy, bxy)]

def plot_roads(df, ax=None):
    from matplotlib.pyplot import gca
    from matplotlib.collections import LineCollection
    if ax is None: ax = gca()
    A = node_coords(df["AX"], df["AY"])
    B = node_coords(df["BX"], df["BY"])
    E = edge_coords(A, B)
    ec = LineCollection(E)
    ax.add_collection(ec)
    ax.autoscale()

def plot_point(x, y, markerstyle):
    from matplotlib.pyplot import plot
    plot(x, y, markerstyle, markersize=10)

def get_intersection_coords(i, df):
    return df["X"].iloc[i], df["Y"].iloc[i]
```

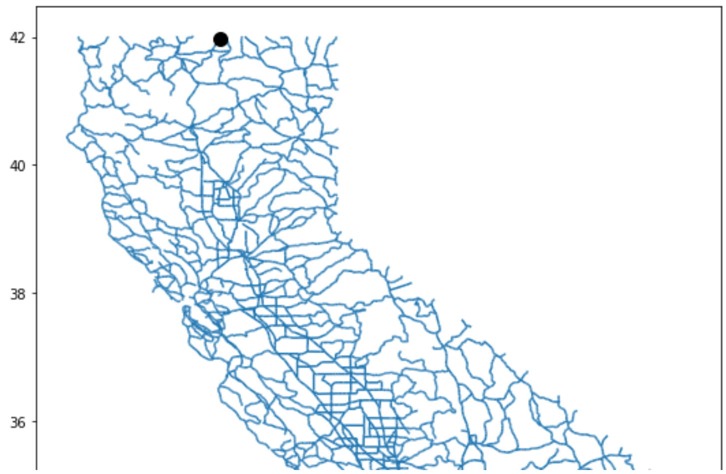
To see this visualization in action, let's load all intersections, and consider the first one (first row) and the last one (last row):

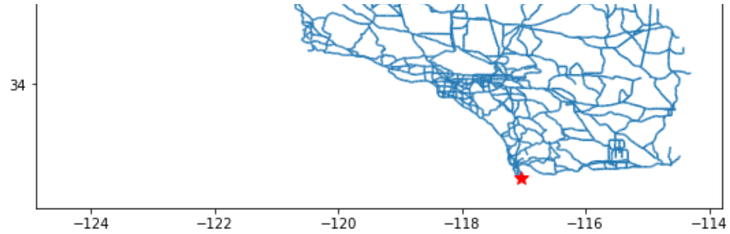
In [10]:
```
df_intersections = pd.read_sql_query("SELECT * FROM Intersections", conn)

i_first = 0
i_last = -1
display(df_intersections.iloc[[i_first, i_last]])
```

	ID	X	Y
0	0	-121.904167	41.974556
21047	21047	-117.035332	32.541302

Let's draw the road network and mark these two intersections:

In [11]:
```
plt.figure(figsize=(9, 9))
plot_roads(df_roads)
plot_point(*get_intersection_coords(i_first, df_intersections), 'ko')
plot_point(*get_intersection_coords(i_last, df_intersections), 'r*')
```





Closing the database. The cells below assume the use of pandas, not SQL. For this reason, the next code cell will close the database connection. If for some reason you think you need to leave it open, then comment out this line (but do try to remember to close the connection later!).

In [12]:
```
conn.close() # Close the SQL database for good measure
```

Exercise 1 (1 point): Calculating road segment lengths

To calculate the shortest path between any two intersections, we need to know the lengths of each road segment.

Let's use the Euclidean distance as our measure of length. That is, let a and b be two intersections whose (x, y)-coordinates are (x_a, y_a) for point a and (x_b, y_b) for point b. Then the Euclidean distance $d(a, b)$ between them is

$$d(a, b) \equiv \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}.$$

Unfortunately, sqlite3 does **not** have an easy built-in way to compute square roots. Therefore, let's switch to using pandas, now that we've extracted the main tables we need.

Complete the function `calc_distances(df)`, below. Assume that the input data frame, `df`, is one generated by your query in Exercise 0, e.g., it has the form,

E	A	AX	AY	B	BX	BY
0	0	-121.90416699999999	41.974556	1	-121.902153	41.974765999999995
1	0	-121.90416699999999	41.974556	6	-121.91008799999999	41.973942
2	1	-121.902153	41.974765999999995	2	-121.89679	41.988075
3	2	-121.89679	41.988075	3	-121.88960300000001	41.998032
4	3	-121.88960300000001	41.998032	4	-121.88668100000001	42.008739

...

Your function should return a new pandas `DataFrame` with just two columns: `E`, holding the road segment ID, and `L`, holding the Euclidean length of that segment. For instance, the result for the rows shown above would be

E	L
0	0.0020249187638055285
1	0.005952750372727481
2	0.01434891110851333
3	0.012279854152229423
4	0.011098555446539574

For instance, consider segment 2. Its length is

$$\sqrt{(-121.89679 - (-121.902153))^2 + (41.988075 - 41.974765999999995)^2)} \approx 0.01434891111\ldots,$$

as indicated above.

In [13]:
```
def calc_distances(df):
    ### BEGIN SOLUTION
    return calc_distances__0(df)

# Solution 0: Basic pandas/Numpy
def calc_distances__0(df):
    from numpy import sqrt
    df_dist = df.copy()
    dx = df_dist["AX"] - df_dist["BX"]
```

```
    dy = df_dist["AY"] - df_dist["BY"]
    df_dist["L"] = sqrt(dx**2 + dy**2)
    return df_dist[["E", "L"]]

# Solution 1: Apply
def calc_distances__1(df):
    df_dist = df.copy()
    df_dist["L"] = df_dist[["AX", "AY", "BX", "BY"]].apply(euclidean_distance, axis=1)
    return df_dist[["E", "L"]]

def euclidean_distance(row):
    from math import sqrt
    dx = row["AX"] - row["BX"]
    dy = row["AY"] - row["BY"]
    return sqrt(dx*dx + dy*dy)
    ### END SOLUTION

df_distances = calc_distances(df_roads)
df_distances.head()
```

Out[13]:

	E	L
0	0	0.002025
1	1	0.005953
2	2	0.014349
3	3	0.012280
4	4	0.011099

In [14]:
```
# Test cell: `ex1__calc_distances` (1 point)

### BEGIN HIDDEN TESTS
def ex1_soln(df):
    from numpy import sqrt
    df_dist = df.copy()
    dx = df_dist["AX"] - df_dist["BX"]
    dy = df_dist["AY"] - df_dist["BY"]
    df_dist["L"] = sqrt(dx**2 + dy**2)
    return df_dist[["E", "L"]]

#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#

def ex1_gen_soln(force=False):
    from problem_utils import get_path
    from os.path import isfile
    from problem_utils import pandas_df_to_markdown_table
    from pandas import read_csv

    soln_file = get_path('ca-roads/ex1_soln.csv')
    if force or not isfile(soln_file):
        print(f"Generating solution file, '{soln_file}'...")
        df_roads = read_csv(get_path('ca-roads/ex0_soln.csv'))
        df_soln = ex1_soln(df_roads)
        df_soln.to_csv(soln_file, index=False)
        print("=== Demo ===")
        print(pandas_df_to_markdown_table(df_soln.head(), index=False))
    else:
        print(f"Solution file, '{soln_file}', already exists; skipping re-generation...")
```

```python
ex1_gen_soln()
### END HIDDEN TESTS

def ex1_gen_df():
    from random import randint, random, sample
    from itertools import combinations
    from pandas import DataFrame
    from math import sqrt

    def euclidean(ax, ay, bx, by):
        return sqrt((bx - ax)**2 + (by - ay)**2)

    num_ints = randint(3, 5)
    ids = list(range(num_ints))
    xs, ys = [], []
    for v in ids:
        xs.append(-1 + 2*random())
        ys.append(-1 + 2*random())
    all_roads = list(combinations(ids, 2))
    num_roads = randint(1, len(all_roads))
    roads = set()
    for a, b in sample(all_roads, num_roads):
        roads |= {(a, b)}
    rr, aa, bb = [], [], []
    for r, (a, b) in enumerate(roads):
        rr += [r]
        aa += [a]
        bb += [b]
    axs = [xs[a] for a in aa]
    ays = [ys[a] for a in aa]
    bxs = [xs[b] for b in bb]
    bys = [ys[b] for b in bb]
    ll = [euclidean(ax, ay, bx, by) for ax, ay, bx, by in zip(axs, ays, bxs, bys)]
    df = DataFrame({"E": rr,
                    "A": aa, "AX": axs, "AY": ays,
                    "B": bb, "BX": bxs, "BY": bys})
    df_soln = DataFrame({"E": rr, "L": ll})
    return df, df_soln

def ex1_check_one():
    df, df_soln = ex1_gen_df()
    try:
        df_your_soln = calc_distances(df)
        assert_tibbles_are_equivalent(df_soln, df_your_soln)
    except:
        print("\n*** ERROR ***\n")
        print("Your code did not produce the expected solution.")
        print("Input data frame:")
        display(df)
        print("Expected solution:")
        display(df_soln)
        print("Your solution:")
        display(df_your_soln)
        raise

print()
for trial in range(10):
    print(f"=== Trial #{trial} / 9 ===")
    ex1_check_one()

print("\n(Passed.)")
```

```
Solution file, './resource/asnlib/publicdata/ca-roads/ex1_soln.csv', already exists; skipping re-generation...

=== Trial #0 / 9 ===
=== Trial #1 / 9 ===
=== Trial #2 / 9 ===
=== Trial #3 / 9 ===
=== Trial #4 / 9 ===
=== Trial #5 / 9 ===
=== Trial #6 / 9 ===
=== Trial #7 / 9 ===
=== Trial #8 / 9 ===
=== Trial #9 / 9 ===

(Passed.)
```

Pre-computed distances

Whether you solved Exercise 1 successfully or not, the following code cell will load a pre-generated solution for these data and store them in a pandas `DataFrame` called `df_distances`, so that you can continue with the problem. Subsequent exercises depend on `df_distances`, so do **not** modify it.

In [15]:
```
df_distances = pd.read_csv(get_path("ca-roads/ex1_soln.csv"))
display(df_distances.head())
print("...")
```

	E	L
0	0	0.002025
1	1	0.005953
2	2	0.014349
3	3	0.012280
4	4	0.011099

```
...
```

Edge lists

To use NetworkX, a *graph* or *network* is a collection of *nodes* (or *vertices*) and *edges*, which connect nodes. For instance, in the road network, intersections are nodes and roads are edges. In the airport network of Notebook 11, airports were nodes and direct flight segments were edges. And in a social network, a person is a node and a friendship connection is an edge.

As a starting step, let's combine the `df_roads` and `df_distances` data frames into a single data frame, called `df_edges`:

In [16]:
```
df_edges = df_roads.merge(df_distances, on="E")
df_edges.head()
```

Out[16]:

	E	A	AX	AY	B	BX	BY	L
0	0	0	-121.904167	41.974556	1	-121.902153	41.974766	0.002025
1	1	0	-121.904167	41.974556	6	-121.910088	41.973942	0.005953
2	2	1	-121.902153	41.974766	2	-121.896790	41.988075	0.014349
3	3	2	-121.896790	41.988075	3	-121.889603	41.998032	0.012280
4	4	3	-121.889603	41.998032	4	-121.886681	42.008739	0.011099

NetworkX requires us to define nodes and edges in a certain way. For nodes, we can just use the intersection IDs as node IDs.

For edges, we need to construct an *edge list*. An edge list is a list of tuples of the form, `(a, b, s)`, where

- `a` represents one node ID of a given edge;
- `b` represents the other node ID of that edge;
- and `s` is a dictionary of attributes (possibly empty).

We'll explain how we'll use `s` in the next exercise, where you'll create an edge list for our data.

Exercise 2 (1 point): Creating an edge list

Complete the function `get_edgelist(df)`, below.

The input data frame, `df`, will be something that looks like the `df_edges` data frame from above, that is, a tibble of road segments with intersection IDs (columns `"A"` and `"B"`), coordinates (`"AX"`, `"AY"`, `"BX"`, `"BY"`), and segment length (`"L"`).

Your function should convert this data frame into a Python list of edge-tuples, $[(a_0, b_0, s_0), (a_1, b_1, s_1), \ldots]$ such that for each edge k,

- (a_k, b_k) are the intersection IDs of road segment k; and
- s_k is a dictionary with exactly one key, `"w"`, whose value is the length of segment k.

For example, for the first five rows of `df_edges` above, the returned list would have the elements

```
[(0, 1, {'w': 0.002025}),
     (0, 6, {'w': 0.005953}),
```

```
 (1, 2, {'w': 0.014349}),
 (2, 3, {'w': 0.012280}),
 (3, 4, {'w': 0.011099}),
 ...]
```

Note 0. The intersection IDs in your solution must be of type `int`.

Note 1. Your solution should not depend on the order of the columns in `df`. The test cell may check your code on data frames where, for instance, `"L"` is the second column or `"L"` is the last column.

In [17]:

```
def get_edgelist(df):
    ### BEGIN SOLUTION
    return [(a, b, {'w': w}) for a, b, w in zip(df["A"], df["B"], df["L"])]
    ### END SOLUTION

# Demo
edgelist = get_edgelist(df_edges)
edgelist[:5]
```

Out[17]:

```
[(0, 1, {'w': 0.0020249187638055285}),
 (0, 6, {'w': 0.005952750372727481}),
 (1, 2, {'w': 0.014348911108513331}),
 (2, 3, {'w': 0.012279854152229423}),
 (3, 4, {'w': 0.011098555446539574})]
```

In [18]:

```
# Test cell: `ex2__get_edgelist` (1 point)

### BEGIN HIDDEN TESTS
def ex2_soln(df):
    return [(a, b, {'w': w}) for a, b, w in zip(df["A"], df["B"], df["L"])]

#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#

def ex2_write_soln(force=False):
    from os.path import isfile
    from pandas import read_csv
    soln_file = get_path("ca-roads/ex2_soln.csv")
    if force or not isfile(soln_file):
        print(f"Generating solution file, '{soln_file}'...")
        df_roads = read_csv(get_path("ca-roads/ex0_soln.csv"))
        df_distances = read_csv(get_path("ca-roads/ex1_soln.csv"))
        df_edges = df_roads.merge(df_distances, on="E")
        soln_list = ex2_soln(df_edges)
        with open(soln_file, "wt") as fp:
            for a, b, s in soln_list:
                fp.write(f"{a},{b},{s['w']}\n")
    else:
        print(f"Solution file, '{soln_file}', exists; skipping...")

ex2_write_soln()
### END HIDDEN TESTS

def ex2_gen_df():
    from random import randint, random, sample
    from itertools import combinations
    from pandas import DataFrame
    from math import sqrt
```

```
    def euclidean(ax, ay, bx, by):
        return sqrt((bx - ax)**2 + (by - ay)**2)

    num_ints = randint(3, 5)
    ids = list(range(num_ints))
    xs, ys = [], []
    for v in ids:
        xs.append(-1 + 2*random())
        ys.append(-1 + 2*random())
    all_roads = list(combinations(ids, 2))
    num_roads = randint(1, len(all_roads))
    roads = set()
    for a, b in sample(all_roads, num_roads):
        roads |= {(a, b)}
    rr, aa, axs, ays, bb, bxs, bys, ll = [], [], [], [], [], [], [], []
    soln_list = []
    for r, (a, b) in enumerate(roads):
        rr += [r]
        aa += [a]
        axs += [xs[a]]
        ays += [ys[a]]
        bb += [b]
        bxs += [xs[b]]
        bys += [ys[b]]
        ll += [euclidean(axs[-1], ays[-1], bxs[-1], bys[-1])]
        soln_list.append((a, b, {'w': ll[-1]}))
    df = DataFrame({"E": rr, "L": ll,
                    "A": aa, "AX": axs, "AY": ays,
                    "B": bb, "BX": bxs, "BY": bys})
    return df, soln_list

def ex2_check_one():
    from math import isclose
    def soln_to_dict(soln):
        assert isinstance(soln, list), f"*** Solution should be a list, not `{type(soln)}`. ***"
        soln_dict = {}
        for k, x in enumerate(soln):
            assert isinstance(x, tuple), f"*** Element {k} == '{x}', which is a `{type(x)}`, not a `{type(tuple())}
`. ***"
            assert len(x) == 3, f"*** Element {k} == '{x}' has {len(x)} values instead of just 3. ***"
            assert isinstance(x[-1], dict), \
                   f"*** For element {k} == '{x}', third component is a `{type(x[-1])}` rather than a `dict`. ***"
            assert 'w' in x[-1], f"*** The dictionary at element {k} == '{x}' does not have a 'w' key. ***"
            soln_dict[(x[0], x[1])] = x[2]
        return soln_dict

    df, soln_list = ex2_gen_df()
    soln_dict = soln_to_dict(soln_list)
    try:
        your_soln_list = get_edgelist(df)
        your_soln_dict = soln_to_dict(your_soln_list)
        for (ura, urb), urs in your_soln_dict.items():
            assert (ura, urb) in soln_dict, f"*** Edge ({ura}, {urb}) is not in the expected solution. ***"
            assert isclose(urs['w'], soln_dict[(ura, urb)]['w']), \
                   f"*** For ({ura}, {urb}), weight {urs['w']} does not match ours, {soln_dict[(ura, urb)]}. ***"
    except:
        print("\n*** ERROR: Your results don't match our expectations. ***\n")
        print("==> Input data frame:")
        display(df)
        print("==> Expected output:")
        print(soln_list)
        print("==> Your output:")
        print(your_soln_list)
        raise

print()
for trial in range(10):
    print(f"=== Trial #{trial} / 9 ===")
    ex2_check_one()

print("\n(Passed.)")
```

```
Solution file, './resource/asnlib/publicdata/ca-roads/ex2_soln.csv', exists; skipping...

=== Trial #0 / 9 ===
=== Trial #1 / 9 ===
=== Trial #2 / 9 ===
=== Trial #3 / 9 ===
=== Trial #4 / 9 ===
=== Trial #5 / 9 ===
=== Trial #6 / 9 ===
```

```
=== Trial #6 / 9 ===
=== Trial #7 / 9 ===
=== Trial #8 / 9 ===
=== Trial #9 / 9 ===

(Passed.)
```

A pre-computed edge list

Whether you solved Exercise 2 or not, the following code cell will load a pre-generated solution for these data and store them in a list called `edgelist`, so that you can continue with this problem. Subsequent exercises depend on `df_distances`, so do **not** modify it.

In [19]:
```
with open(get_path("ca-roads/ex2_soln.csv"), "rt") as fp:
    edgelist = []
    for l in fp.readlines():
        a, b, s = l.strip().split(',')
        edgelist.append((int(a), int(b), {'w': float(s)}))

print("First five edges:")
for e in edgelist[:5]:
    print(e)
print("...")
```

```
First five edges:
(0, 1, {'w': 0.0020249187638055285})
(0, 6, {'w': 0.005952750372727481})
(1, 2, {'w': 0.014348911108513331})
(2, 3, {'w': 0.012279854152229423})
(3, 4, {'w': 0.011098555446539574})
...
```

Computing shortest paths via NetworkX

At last, we are ready to calculate shortest paths!

First, let's create a NetworkX `Graph` object with intersections as nodes and the edge list from Exercise 2.

> By default, NetworkX treats the graph as *undirected*. So edge (0, 6) and (6, 0) are treated as the same.

In [20]:
```
from networkx import Graph
G = Graph()
G.add_nodes_from(df_intersections["ID"])
G.add_edges_from(edgelist)

print(f"The graph has {G.number_of_nodes()} nodes and {G.number_of_edges()} edges.")
```

```
The graph has 21048 nodes and 21693 edges.
```

Once we have a graph, asking for a shortest path between two nodes is easy! First, recall that each element of our edge list had two end points, a and b, as well as the distance between them. When we build the graph, we will ask NetworkX to use the distances as *weights* along each edge. Then, when searching for a shortest path, NetworkX will use these edge weights to calculate the length of a path, and the shortest path will be the one where the sum of edge weights is minimized.

The following code uses NetworkX to compute the path between node (intersection) `start` and `finish`:

In [21]:
```
def get_shortest_path(s, t, G):
    from networkx import shortest_path
    return shortest_path(G, source=s, target=t, weight='w')

start = df_intersections["ID"].iloc[i_first]
finish = df_intersections["ID"].iloc[i_last]
print(f"Calculating a shortest path between nodes (intersections) {start} and {finish}...")
path = get_shortest_path(start, finish, G)

print(f"\n==> First ten nodes (intersections) along the path, which is of type `{type(path)}`:")
print('    ' + ', '.join([str(s) for s in path[:10]]) + ", ...")
```

```
Calculating a shortest path between nodes (intersections) 0 and 21047...

==> First ten nodes (intersections) along the path, which is of type `<class 'list'>`:
    0, 6, 5, 7, 265, 264, 263, 262, 261, 260, ...
```

The `path` is a list of nodes (intersections). For instance, if the path were the list,

```
[8, 2, 6, 3, 10, 11, 104, 52]
```

then that would mean the shortest path starts at node 8, then goes along the edge from 8 to 2, then along the edge from 2 to 6, and so on.

Exercise 3 (2 points): Edge coordinates, for plotting

As your last exercise, let's take a path and determine the coordinates of each edge, producing an *edge coordinates list*. It's easiest to see by example.

Example. Suppose the coordinates of these nodes is given in an intersections data frame, `df`, which is the following:

ID	X	Y
0	-121.90416699999999	41.974556
5	-121.915062	41.970314
6	-121.91008799999999	41.973942
7	-121.916199	41.969482

Now suppose you are given the following path between nodes 0 and 7: `path = [0, 6, 5, 7]`.

The *edge coordinates list* of this path is a list with one element per edge of the path. In this example, the edges of the path are (0, 6), (6, 5), and (5, 7), so the edge coordinates list will have three elements. Each element is also a list. Each inner list has two elements, which are two (x, y)-coordinate pairs. In our example.

```
[[(-121.90416699999999, 41.974556), (-121.91008799999999, 41.973942)], # (x, y) for 0 and 6
    [(-121.91008799999999, 41.973942), (-121.915062, 41.970314)], # (x, y) for 6 and 5
    [(-121.915062, 41.970314), (-121.916199, 41.969482)] # (x, y) for 5 and 7
   ]
```

Your task. Complete the function, `path_to_coords(path, df)`, so that it computes the edge coordinates list for a path, `path`, whose node coordinates are given by `df`. That is, `path` will be a list of node IDs and `df` will hold the coordinates where column `"ID"` holds the node ID, `"X"` the x-coordinate, and `"Y"` the y-coordinate. It should return this edge coordinates list.

In [22]:
```python
def path_to_coords(path, df):
    ### BEGIN SOLUTION
    df = df.set_index("ID")
    edgecoords = []
    for a, b in zip(path[:-1], path[1:]):
        ax, ay = df["X"].loc[a], df["Y"].loc[a]
        bx, by = df["X"].loc[b], df["Y"].loc[b]
        edgecoords.append([(ax, ay), (bx, by)])
    return edgecoords
    ### END SOLUTION

print("Converting the path to coordinates...")
path_coords = path_to_coords(path, df_intersections)
print("The first five edge coordintes along the shortest path:")
path_coords[:5]
```

```
Converting the path to coordinates...
The first five edge coordintes along the shortest path:
```

Out[22]:
```
[[(-121.90416699999999, 41.974556), (-121.91008799999999, 41.973942)],
 [(-121.91008799999999, 41.973942), (-121.915062, 41.970314)],
 [(-121.915062, 41.970314), (-121.916199, 41.969482)],
 [(-121.916199, 41.969482), (-121.918793, 41.967587)],
 [(-121.918793, 41.967587), (-121.946999, 41.921818)]]
```

In [23]:
```python
# Test cell: `ex3__path_to_coords` (2 points)

def ex3_gen():
    from random import randint, random, sample
    from pandas import DataFrame

    num_ints = randint(2, 10)
    ids = list(range(num_ints))
    xs, ys = [], []
    for v in ids:
        xs.append(-1 + 2*random())
        ys.append(-1 + 2*random())
```

```
    path_len = randint(2, num_ints)
    path = sample(ids, path_len)
    soln = [[(xs[i], ys[i]), (xs[j], ys[j])] for i, j in zip(path[:-1], path[1:])]
    df = DataFrame({"ID": ids, "X": xs, "Y": ys})
    return path, df, soln

def ex3_check_one():
    from math import isclose
    def check_types(soln):
        assert isinstance(soln, list), "f*** Solution should be a `list`, not a `{type(soln)}`. ***"
        for k, e in enumerate(soln):
            assert isinstance(e, list), f"*** Element {k} (== {e}) should be a `list`, not a `{type(e)}`. ***"
            assert len(e) == 2, f"*** Element {k} (== {e}) should have two elements, not {len(e)}. ***"
            assert isinstance(e[0], tuple), \
                   f"*** First part of element {k} (== {e}) should be a `tuple`, not a `{type(e[0])}`. ***"
            assert isinstance(e[1], tuple), \
                   f"*** Second part of element {k} (== {e}) should be a `tuple`, not a `{type(e[1])}`. ***"
            assert len(e[0]) == 2, \
                   f"*** First part of element {k} (== {e}) should be a pair, not a {len(e[0])}-tuple. ***"
            assert len(e[1]) == 2, \
                   f"*** First part of element {k} (== {e}) should be a pair, not a {len(e[1])}-tuple. ***"
    def check_solns(yours, soln):
        check_types(your_soln)
        assert len(yours) == len(soln), \
               f"*** Your solution has {len(yours)} elements, whereas we expected {len(soln)}. ***"
        for k, (u, x) in enumerate(zip(yours, soln)):
            for i in range(2):
                for j in range(2):
                    assert isclose(u[i][j], x[i][j]), \
                           f"*** Mismatch at position {k}: yours is {u}, whereas we expected {x}. ***"

    path, df, soln = ex3_gen()
    check_types(soln)
    try:
        your_soln = path_to_coords(path, df)
        check_solns(your_soln, soln)
    except:
        print("\n*** ERROR: Your results don't match our expectations. ***\n")
        print("==> Input path:", path)
        print("==> Input data frame:")
        display(df)
        print("==> Expected output:")
        print(soln)
        print("==> Your output:")
        print(your_soln)
        raise

print()
for trial in range(10):
    print(f"=== Trial #{trial} / 9 ===")
    ex3_check_one()

print("\n(Passed.)")
```

```
=== Trial #0 / 9 ===
=== Trial #1 / 9 ===
=== Trial #2 / 9 ===
=== Trial #3 / 9 ===
=== Trial #4 / 9 ===
=== Trial #5 / 9 ===
=== Trial #6 / 9 ===
=== Trial #7 / 9 ===
=== Trial #8 / 9 ===
=== Trial #9 / 9 ===

(Passed.)
```

Putting it all together: visualizing the shortest path

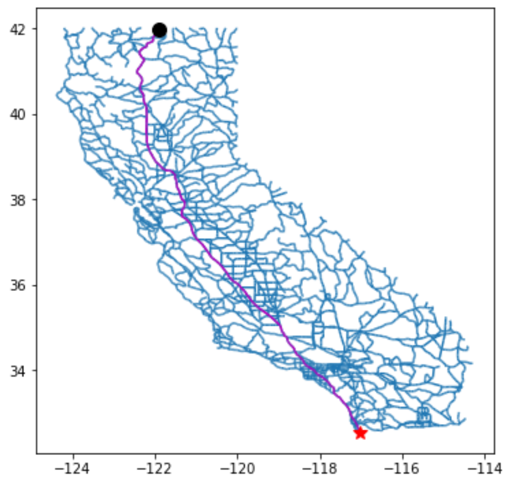
If your code works, then the following will create a visualization identical to the one at the very top of this notebook.

There are no more exercises in this problem, so if you believe you are done, the rest is optional.

In [24]:
```
# This cell will plot a route. Feel free to edit the
# starting and finishing nodes to see other paths!

def plot_edgecoords(edgecoords, ax=None, color=None):
    from matplotlib.pyplot import gca
```

```
    from matplotlib.collections import LineCollection
    if ax is None: ax = gca()
    if color is None: color = ['m'] * len(edgecoords)
    ec = LineCollection(edgecoords, color=color)
    ax.add_collection(ec)

# Modify `start` and `finish`, if you want to see other paths!
# For example, uncomment the lines that pick random
# starting and ending nodes.
start = df_intersections["ID"].iloc[i_first]
finish = df_intersections["ID"].iloc[i_last]

#start = int(df_intersections["ID"].sample(1)) # Random start
#finish = int(df_intersections["ID"].sample(1)) # Random finish

print(f"Calculating a shortest path between nodes (intersections) {start} and {finish}...")
path = get_shortest_path(start, finish, G)
path_coords = path_to_coords(path, df_intersections)

plt.figure(figsize=(6, 6))
plot_roads(df_roads)
plot_point(*get_intersection_coords(start, df_intersections), 'ko')
plot_point(*get_intersection_coords(finish, df_intersections), 'r*')
plot_edgecoords(path_coords)
```

```
Calculating a shortest path between nodes (intersections) 0 and 21047...
```



Fin! You've reached the end of this problem. Don't forget to restart and run all cells again to make sure it's all working when run in sequence; and make sure your work passes the submission process. Good luck!