The Traveling Salesman Problem: National Parks Edition

Naomi Dailey | ABT 182 Final Project

Overview

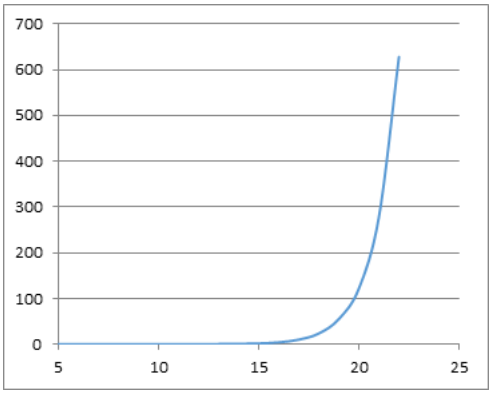
The Traveling Salesman Problem (TSP) is one of the most famous problems in computer science. It analyzes the age-old question of determining optimal spatial paths between many locations (Hoffman et al., 2013). Specifically, the TSP asks:

> *Given a list of points and the distances between them, what is the shortest path that visits each city once and returns to the origin?*

The TSP is based on the historical traveling salesman of Willy Loman fame. Individuals traveled door-to-door, enticing residents to purchase Cutco knives, Tupperware, vaccuums, and the like. Salesmen like Willy had to carefully plan an efficient route from city to city--the shorter the route the better.

This conundrum seduced computer scientists because of how computationally expensive it is. To search for the optimal path for all possibilities among *n* points is more difficult the more points you have. In computer science, the TSP is classified as an *O(n!)* algorithm (Google Developers, 2015). In essence, *n!* computational steps are needed to find a solution, which rapidly increases the computing time. Thus, an automation problem was born. The images below are from Mihailo Lalevic's brute force solution of the TSP, showing how dramatically processing time increases with additional points.

Points	Time (seconds)
5	0
6	0
7	0.001
8	0.002
9	0.007
10	0.019
11	0.045
12	0.114
13	0.291
14	0.705
15	1.72
16	4.152
17	9.995



Problem

Instead of the shortest distance between cities, this project will enable us to find the shortest distances between any points with longitude and latitude values. In the example presented here, we use coordinates for U.S. National Parks scraped from Wikipedia, but you could substitute these with points scraped from any web page (given that you know the HTML tags). Alternatively, we can skip the web scraping and use a CSV that contains coordinates, also outlined below (Option 2).

You may be thinking, *doesn't Google Maps already do this?* The answer is yes, but only to a point (pun intended). Google Maps Directions API provides superb applications for paths between waypoints, but limits the user to 23 such points. This leaves much to be desired if we have a long list of destinations we want to hit, like this epic National Park road trip. By inputting user-specified points with associated coordinate values, this project returns the optimal itinerant route via Euclidean distance.

Concepts

While the TSP is relatively easy to describe, it remains a delightful thought experiment because it is so computationally expensive to solve. Hoffman et al. (2013) describe the TSP as part of a larger group of problems that are known as combinatorial optimization problems. Combinatorial optimization finds an optimal object when an exhaustive search is not feasible. The set of feasible solutions is discrete (or must be reduced to a discrete set). Beyond thought experiments like the TSP, combinatorial optimization is an integral component of applications like artificial intelligence and machine learning.

Brute Force

To solve a problem like the TSP by brute force isn't really feasible. Mihailo Lalevic wrote a Python script that shows the relationship between number of points and processing time; when you reach 22 points, the performance declines immensely because of memory usage (and takes over 10 minutes!). If I were to try to solve my National Parks problem by brute force, the number of permutations of possible routes is staggering:

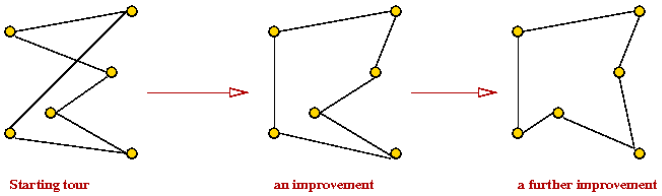
```
import math
math.factorial(44)

2658271574788448768043625811014615890319638528000000000
```

That's a lot of potential routes. It's also worth noting that if we were to assume 1000 combinations analyzed per second, this TSP would take 8.429e43 years to complete. So maybe brute force is not the best solution for our road trip--we'd like to spend more time traveling than planning potential routes!

Christofides Algorithm

Fortunately, others have created computer programs and algorithms to make the TSP easy for amateurs like myself. Many solutions to the TSP reference Christofides algorithm (1976), which finds approximate solutions to the TSP based on instances where distances are symmetric and obey the triangle equality concept. This algorithm guarantees that its solutions will be within a factor of 1.5 of the optimal solution length.



Branch-and-Bound Algorithm

Mortada Meyhar found that the Concorde computer program for the TSP (Version 3.12.19, University of Waterloo) could find the exact optimal path between Tesla supercharger stations, without approximation. Concorde does this using branch-and-bound algorithms: "when the algorithm branches out to search for the optimum, many of the permutations can actually be safely cut short if it is impossible for a branch to result in a value better than a known better solution" (Mehyar, 2015). Mehyar has created a helpful example of the optimal driving path between Tesla supercharger stations, built with the Google Maps Directions API.

Solving the TSP

We'll be emulating Meyhar's method to find the optimum path (as the crow flies) through our National Park coordinates. Below we'll go through each step, showing the code and outputs, and try to make the script as generalizable as possible. Please note that this markdown file and the script were written using the iPython Notebook, so there are some magic commands at the top of some of the cells. Additionally, this project was written in Python 3.5, which may cause problems if running other versions of Python (many modules like `numpy` and `pandas` are version-specific).

Other helpful tools to solve the TSP and create web maps included:

- Concorde (mac installations here or '/Dailey_FinalProject/README.pdf')

- QSOPT (mac installations here)
- Leaflet (install directions here)
- Command line (great tutorial here)

Set-Up

Below are the modules necessary to complete this script, as well as a simple folder structure to organize the project. Please note that some of the modules (e.g., `pandas` and `numpy`) require command line installation using `pip install`. Also, don't worry about the `map_oldpath` and `map_newpath` just yet; this is so our web map .html document is in the right directory for the final product.

Please place the '/Dailey_FinalProject' folder on your desktop so the working directory is defined correctly.

Import modules and set directory

```
import bs4, re, requests, json, os, csv
import matplotlib.pyplot as plt
import numpy as npy
import pandas as pd
from geojson import Feature, Point, FeatureCollection, LineString
from shapely.geometry import Point, mapping

# set up working directory and results path
path = os.path.join(os.getcwd(), "Desktop/Dailey_FinalProject")
outpath = os.path.join(path, "Results")
if not os.path.exists(outpath): os.mkdir(outpath)

# moves the map .html document into your results folder
map_oldpath = os.path.join(path, "TSP_webmap.html")
map_newpath = os.path.join(outpath, "TSP_webmap.html")
os.rename(map_oldpath, map_newpath)
```

Getting Latitude + Longitude

There are two options here for getting the latitude and longitude inputs: web scraping or importing .csv files. First, I'll go through the process of web scraping, which is of particular interest to me. Second, I will provide a way for the user to import her own .csv file as a dictionary of coordinate pairs.

Option 1: Web scraping

Before this project, web scraping data appeared to be a mystery, fit only for the most talented programmers. Enter: BeautifulSoup. This is a fantastic web scraping library that makes it easy to identify and parse your data from HTML source code. (Note that for Mac, I had to enable the Safari Web Inspector to access the source code for a web page.) We will have to scour the source code for tags that matched the coordinates we want, but the Web Inspector makes it easy to identify which HTML tags to look for.

For the record, our method of web scraping changes depending on the website we visit. Wikipedia HTML *tends* to be structured similarly between pages, but the tags we use and geospatial format of our points will change depending on the web page. For example, we can easily fetch latitude and longitude coordinates for the tallest peaks in the U.S. using the methods outlined below, but if we want to scrape Peet's street addresses for a cup of coffee, the tags and geocoding will be different.

Getting the URL (and the source code) is a relatively simple affair. Getting the right spatial reference for the points (e.g., coordinates, addresses) is a different matter. Here we use the Wikipedia page for U.S. National Parks, import the source code as text, and create a BeautifulSoup object that we can use to search for geospatial points.

Scrape via HTML Tags

```
# write function to scrape HTML
def webScrapeAllHTML(myURL):
    page = requests.get(myURL)
    content = page.text  # imports the source code as text
    return (bs4.BeautifulSoup(content, 'lxml'))
    # the bs4 object allows you to search text by tags

# fetch URL from Wikipedia National Parks website
parks_url = 'https://en.wikipedia.org/wiki/List_of_national_parks_of_the_United_States'
soup = webScrapeAllHTML(parks_url)
```

Next, we create a list of latitude and longitude values. We can identify the coordinates from the `soup` object using tags found in the HTML source code. This requires some tedious reading of HTML script to find the tags that point to our coordinates. Extracting the coordinates and appending them to a list requires a regular `for` loop. Putting the coordinates into a list allows us to easily put them in a dictionary later.

Create list of coordinates

```
# identify coordinates using the tags in the webScrapeAllHTML() output
html_coords = soup.find_all("span", "geo")

# extract coordinates from soup and make into a list
coords = []

for tag in html_coords:
    lat, lon = tag.text.strip().split(';')  # pulls coordinates out
    lat, lon = float(lat), float(lon)  # reformat to floats
    latlon = [lat, lon]
    coords.append(latlon)  # create list of lat lon values
```

Now that we have a list of coordinates, we can extract the corresponding National Park names for each coordinate pair. Here we use the same methods as above, but with different HTML tags to extract names and create a list. We will combine the names and coordinates in a dictionary later on.

Create list of National Park names

```
# same as above, but identify names using tags in the webScrapeAllHTML() output
html_names = soup.find_all('th', attrs = {'scope':'row'})

# extract names from soup and make into a list
names = []

for name in html_names:
    extract = name.text.strip()
    names.append(extract)
```

Option 2: Importing CSVs

A CSV can also be used for this process. A `sample.csv` is included in the `Dailey_FinalProject` folder to test this feature. CSVs allow for more flexibility in terms of data entry, and different types of points can be used (e.g., a mix of restaurants and coffee shops instead of only national parks). However, the script below assumes that the .csv file is in the following format:

Name	Latitude	Longitude
Yosemite	37.51	-119.33
Zion	37.12	-112.59

Create coordinate dictionary from CSV

```
# operates under assumption that your .csv is in the current working directory (path)
mycsv = 'sample.csv'
newcsv = 'sample_new.csv'
csvpath = os.path.join(path, mycsv)  # set path to user's .csv

with open(csvpath, mode = 'r') as infile:
    reader = csv.reader(infile)  # read the user's .csv

    with open(newcsv, mode = 'w') as outfile:
        writer = csv.writer(outfile)

        # compile .csv rows into dictionary using dictionary comprehension
        mydict = {rows[0]: [float(rows[1]), float(rows[2])] for rows in reader}

print(mydict.keys())


dict_keys(['Yosemite', 'Acadia', 'Badlands', 'Zion', 'Arches'])
```

Data Formatting

Now that we have a list of coordinates and a list of park names, we need to combine them in a format that is easy to reference. Fortunately, the pandas library is a great option for building and manipulating data frames.

First, we put the names and coordinates into a dictionary. This makes it easier to eliminate parks that are not in the contiguous United States (which would make our road trip a bit more difficult...cars don't mix well with water). Using the dictionary, we are able to make a rough bounding box with longitude values to limit our parks. For National Parks that fall within our longitude range, but are not in the contiguous U.S., we can take a shortcut and manually delete them (e.g., Channel Islands and Dry Tortugas).

NOTE: If you used the .csv option to import your data, you can skip this step!

Build dictionary and delete unwanted parks

```
# make a dictionary of names with associated coordinates
NPdict = dict(zip(names, coords))

# identify parks outside of contiguous US
keys_to_remove = []

for key in NPdict.keys():
    if (NPdict[key][1] < -130 or NPdict[key][1] > -65):
        # append those identified parks into a list
        keys_to_remove.append(key)

# delete unwanted parks that lie outside of contiguous US
for key in keys_to_remove:
    NPdict.pop(key, None)

# delete stragglers (island parks that fall within the bounding box)
del NPdict['Channel Islands']
del NPdict['Dry Tortugas']
```

Finally, we can make a nicely formatted data frame for our viewing pleasure. Coordinates are indexed by the National Park name, and sorted into latitude and longitude columns. If data was imported from the CSV option, we can simply replace `from_dict(NPdict)` in the first line with the name of our dictionary: `from_dict(mydict)`.

Use dictionary to create data frame

```
# make data frame from dictionary
dataframe = pd.DataFrame.from_dict(NPdict).T
dataframe.columns = ['Lat', 'Lon']
dataframe.head()
```

	Lat	Lon
Acadia	44.35	-68.21
Arches	38.68	-109.57
Badlands	43.75	-102.50
Big Bend	29.25	-103.25
Biscayne	25.65	-80.08

Computing Distances

Computing geodesic distances on Earth's surface is easy enough. The function below is replicated from John D. Cook's very clear Python script on his blog. This function calculates the distance between Point 1 and Point 2 on the surface of the earth, defined by the points' latitude and longitude. Rather than calculating distance on a flat plane, this function computes spherical distances from the converted coordinates, giving us a more accurate represetation of distances.

Write distance function

```
# calculate geodesic distances for any set of lat lon values you have
# NOTE: this calculates Euclidean distance, not driving distance!!

def earthDistance(lat1, lon1, lat2, lon2, radius = 6378.388, unit_miles = False):
    """
    This equation computes distances between two points on Earth's
    surface specified by lat and lon.
    Assume Earth to be a perfect sphere with a given radius (default is 6378.388 km).
    Function can output miles or km (default is km).

    Reference
    ---------
    Adapted from John D. Cook's blog post: http://www.johndcook.com/blog/
    python_longitude_latitude/
    and from Mortada Mehyar's blog post: http://mortada.net/the-traveling-tesla-
    salesman.html
    """
    # convert lat and lon to spherical coordinates in radians
    deg_to_rad = npy.pi / 180.0

    # phi = 90 - lat
    phi1 = (90.0 - lat1) * deg_to_rad
    phi2 = (90.0 - lat2) * deg_to_rad

    # theta = longitude
    theta1 = lon1 * deg_to_rad
    theta2 = lon2 * deg_to_rad

    # compute spherical distance from converted coordinates
    cos = (npy.sin(phi1) * npy.sin(phi2) * npy.cos(theta1 - theta2) + npy.cos(phi1) * npy.cos(phi2))
    arc = npy.arccos(cos)
    length = arc * radius

    if (unit_miles == True):
        length = length/1.60934
    return length
```

We can now use this function to calculate distances between each National Park, in miles. The function will step through a list of National Park names, match the name to the dataframe with latitude and longitude values, and calculate the distances. For example, it will begin with the first park (e.g., Acadia) and calculate the distances between Acadia and all other 43 parks. It will then move to the second park (e.g., Arches) and calculate the distances between Arches and 42 other parks. And so on, until the end of the list is reached.

Use earthDistance() to calculate National Park distances

```
# make list of park names from dictionary to use in distance calculation
dictnames = list(NPdict.keys())
distances = {}

# calculate distances between each national park
for i in range(len(dictnames)):
    # start point
    start = dictnames[i]
    distances[start] = {}

    for j in range(len(dictnames)):
        # stop point
        stop = dictnames[j]

        # the distance is 0 from a point to itself
        if j == i:
            distances[start][stop] = 0.

        # if the names are different, then the earthDistance() function is used
        elif j > i:
            # list names are used to index the lat lon values from the dataframe
            distances[start][stop] = earthDistance(dataframe.ix[start, 'Lat'], dataframe.ix[start, 'Lon'], dataframe.ix[stop, 'Lat'],
dataframe.ix[stop, 'Lon'], unit_miles = True)

        # make sure you don't compute same distances twice
        else:
            distances[start][stop] = distances[stop][start]
```

Data Visualization

Now that we've calculated distances (in miles) between each National Park, we'll do some data visualization before proceeding. The `pandas` dataframe is a great way to check out some of the values, and determine if the results are what we would expect.

Create data frame of earthDistance() output

```
# save distances to new dataframe and view values (in miles)
distances = pd.DataFrame(distances) / km_to_mile
distances.head()
```

	Acadia	Arches	Badlands	Big Bend	Biscayne	Black Canyon of the Gunnison	Bryce Canyon	Canyonlands	Ca
Acadia	0.000000	2154.344980	1692.786260	2180.863111	1454.348997	2066.097994	2315.490775	2185.810886	2247.004
Arches	2154.344980	0.000000	507.911668	745.807969	1935.748550	100.264670	161.443975	38.508500	92.8289(
Badlands	1692.786260	507.911668	0.000000	1003.871836	1775.768221	449.564958	663.098903	545.424276	593.0628
Big Bend	2180.863111	745.807969	1003.871836	0.000000	1441.467071	693.638113	771.928947	728.172291	767.9995
Biscayne	1454.348997	1935.748550	1775.768221	1441.467071	0.000000	1839.158258	2049.861574	1943.816353	2008.254

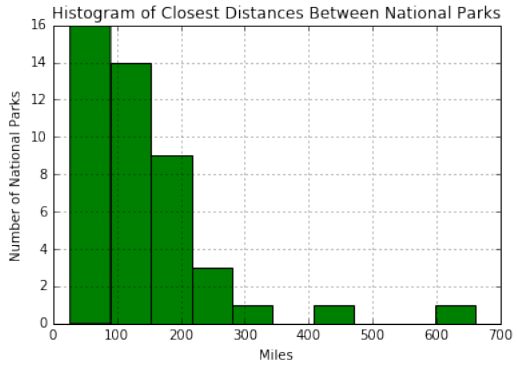
5 rows × 45 columns

Some graphs would also be useful to understand the distribution of distances between parks. Using the `matplotlib` library, we can create some histograms that show the distribution of National Parks that are nearest and furthest from each other.

Create histograms from distance output

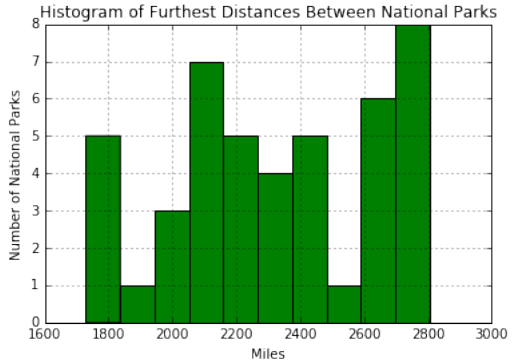
```
# data visualization of distance results
closestNP = distances[distances > 0].min()
cnp = closestNP.hist(bins = 10, color = 'green') # set bins
cnp.set_title('Histogram of Closest Distances Between National Parks') # set labels
cnp.set_ylabel('Number of National Parks')
cnp.set_xlabel('Miles')

<matplotlib.text.Text at 0x118e2fa20>
```




```
furthestNP = distances[distances > 0].max()
fnp = furthestNP.hist(bins = 10, color = 'green')
fnp.set_title('Histogram of Furthest Distances Between National Parks')
fnp.set_ylabel('Number of National Parks')
fnp.set_xlabel('Miles')

<matplotlib.text.Text at 0x1194b2c50>
```



Finding the Optimal Path

Here's the tricky part when the code is no longer, er, *technically* reproducible as is. We are now ready to write our points to a file that can be used by the Concorde TSP Solver. Recall from the '/Dailey_FinalProject/README.pdf' that installing Concorde requires some command line work, and also requires that we have a working C compiler and files from QSOPT. (If you have not done so, I recommend reading the README.pdf) Before using Concorde, we have to install Xcode for Mac and ensure that the command line tools package was downloaded. (I recommend homebrew for these tasks.)

The code below outlines the file format used by Concorde. We first iterate through the list of national park names. Names that match corresponding coordinates get assigned a unique ID number. This output is the `NODE_COORD_SECTION` of the file, and the number of nodes is the `DIMENSION` of the file. The resulting .tsp file is saved in our current directory.

Write .tsp file for Concorde TSP solver

```
# set up empty variables for ID value and file output
node_id = 0
output = ''

# iterate over park names to make an array of ID, latitude and longitude values
for name in dictnames:
    output += '%d %f %f\n' % (node_id, dataframe.ix[name, 'Lat'], dataframe.ix[name, 'Lon'])
    node_id += 1

# format of .tsp file
header = """NAME : NP TSP
COMMENT : National Park TSP
TYPE : TSP
DIMENSION : %d
EDGE_WEIGHT_TYPE : GEOM
NODE_COORD_SECTION
""" % (node_id)

# write .tsp file
tspfile = os.path.join(outpath, 'nationalparks.tsp')

with open(tspfile, 'w') as output_file:
    output_file.write(header)
    output_file.write(output)
```

After running the code above, there will be a .tsp file in '/Dailey_FinalProject/Results/nationalparks.tsp'. If we open the file in a text editor, the output should look like this:

nationalparks.tsp

```
NAME : NP TSP
COMMENT : National Park TSP
TYPE : TSP
DIMENSION : 45
EDGE_WEIGHT_TYPE : GEOM
NODE_COORD_SECTION
0 46.850000 -121.750000
1 32.170000 -104.440000
2 37.180000 -108.490000
3 36.240000 -116.820000
4 38.200000 -109.930000
5 43.750000 -102.500000
6 25.650000 -80.080000
7 48.100000 -88.550000
8 25.320000 -80.930000
9 33.780000 -80.780000
10 43.570000 -103.480000
11 37.730000 -105.510000
12 37.300000 -113.050000
13 33.790000 -115.900000
14 38.680000 -109.570000
15 34.510000 -93.050000
16 35.070000 -109.780000
17 36.430000 -118.680000
```

Execute Concorde

To calculate the optimal route between points, we have to execute Concorde with the .tsp file above. This is easy to do using the command line code below, inputting the username for `<name>` on Mac OS. (NOTE: File path will change for Windows OS)

1. Change the directory to the Concorde TSP folder:

```
cd /Users/<name>/Desktop/Dailey_FinalProject/concorde/TSP
```

2. Execute the Concorde program with the full path to the .tsp file:

```
./concorde /Users/<name>/Desktop/Dailey_FinalProject/Results/nationalparks.tsp
```

After running Concorde, our result appears in the terminal (see image below) and the `nationalparks.sol` result file is saved to the the `Concorde/TSP folder`. In order to manipulate and use the results in a web map, the .sol file must be parsed and read in Python.

```
resnet-108-118:~ naomireimer$ cd /Users/naomireimer/Desktop/Python/concorde/TSP
resnet-108-118:TSP naomireimer$ ./concorde /Users/naomireimer/nationalpark.tsp
./concorde /Users/naomireimer/nationalpark.tsp
Host: resnet-108-118.ucdavis.edu  Current process id: 2510
Using random seed 1457628688
Problem Name: NP TSP
National Park TSP
Problem Type: TSP
Number of Nodes: 45
Geographical Norm in Meters (CC_GEOM)
Set initial upperbound to 17637138 (from tour)
  LP Value  1: 16319971.000000  (0.00 seconds)
  LP Value  2: 17637138.000000  (0.00 seconds)
New lower bound: 17637138.000000
Final lower bound 17637138.000000, upper bound 17637138.000000
Exact lower bound: 17637138.000000
DIFF: 0.000000
Final LP has 52 rows, 141 columns, 358 nonzeros
Optimal Solution: 17637138.00
Number of bbnodes: 1
Total Running Time: 0.03 (seconds)
resnet-108-118:TSP naomireimer$
```

Parse the 'nationalparks.sol' file

```
# import the  nationalparks.sol file
solution_file = os.path.join(path, 'concorde/TSP/nationalparks.sol')

# parse the .sol file to get solutions (list of ID's that correspond to the order of national parks you would visit)
solution = []
f = open(solution_file, 'r')

for line in f.readlines():
    tokens = line.split()
    solution += [int(c) for c in tokens]

f.close()

# ensure the solution length is the same as the park names list length
assert solution[0] == len(dictnames)

# first number in solution is just the dimension (# of nodes), don't need it
solution = solution[1:]

# check that lengths are the same
assert len(solution) == len(dictnames)
```

Now that the solutions have been parsed, we can match each national park's unique ID to its corresponding name. We can then append the solutions into a list and check out the order in which we will be visiting the national parks! We should also double check that we are starting and stopping at the same national park, by changing the list into a `Series` and checking the `head()` and `tail()`.

Append solutions and check optimal path

```
# append solutions into a list
optimal_path = []
for solution_id in solution:
    optimal_path.append(dictnames[solution_id])

# make sure the path starts and ends at the same national park
optimal_path.append(dictnames[solution[0]])

# reformat into a series for cleaner visualization
optimal_path_series = pd.Series(optimal_path)

# check that the first and last national parks are the same
optimal_path_series.head()

0       Mammoth Cave
1    Cuyahoga Valley
2         Shenandoah
3             Acadia
4        Isle Royale
dtype: object

optimal_path_series.tail()

41               Everglades
42                  Biscayne
43                  Congaree
44    Great Smoky Mountains
45             Mammoth Cave
dtype: object
```

We can also calculate the total distance of our road trip (except that this is Euclidean distance, so it would actually be our flight path between national parks. Highway distances is a whole other bag of tricks...)

Calculate total route distance for optimal path

```
# find the total distance of the optimal path
total_dist = 0

for i in range(len(optimal_path) - 1):
    total_dist += distances.ix[optimal_path[i], optimal_path[i + 1]]

total_dist

# total distance should be 10959.222 miles

10959.222975372009
```

Web Mapping with Leaflet

We successfully found the most efficient route and how many miles our trip will be! The next step is to view the trip on a map. Here we'll use Leaflet because open source is the bomb (but there are other web map options out there). Leaflet enables us to use background maps from the awesome cartographers at Stamen, OpenStreetMap, and others.

Leaflet uses JavaScript to create and manipulate web maps. There is a great Codecademy tutorial for JS that will leave you more than capable of writing a script that displays our map. Before writing the Leaflet script, however, we have to put the ordered national park names and their coordinates into a format that can be used in JS. Enter: GeoJSON.

Writing a GeoJSON File

There are some very helpful tools out there to write a GeoJSON file from a Python dictionary. We can create the `tsp_point.json` and the `tsp_line.json` files in two different ways:

1. Using `for` loop, which is great for specific manipulation of properties
2. Using the `geojson` Python module, a quick and dirty version

Option 1: for loop

To structure the `for` loop, we will use this script as a template. It helps structure our Python list of coordinates into a nicely formatted FeatureCollection for our GeoJSON file. It's a bit easier to manipulate the properties of our .json file using this method, which will produce nice informational pop-ups in the final map.

Use for loop to extract coordinates and write to .json

```
# extract coordinates and names from the pandas dataframe
TSP_coords = []

for name in optimal_path:
    TSP_coords += [[dataframe.ix[name, 'Lon'], dataframe.ix[name, 'Lat'], name]]

#gjson will be the main dictionary converted to .json format
gjsonpoint_dict = {}
gjsonpoint_dict["type"] = "FeatureCollection"
feat_list = []

# loop through all points, building a list entry which is a dictionary
for coords in TSP_coords:

    # each of these dictionaries has within it nested a type dict,
    # which contains a point dict and properties dict
    type_dict = {}
    point_dict = {}
    prop_dict = {}

    type_dict["type"] = "Feature"
    point_dict["type"] = "Point"
    type_dict["geometry"] = mapping(Point(coords[0],coords[1]))

    prop_dict["Name"] = coords[2]
    type_dict["properties"] = prop_dict
    feat_list.append(type_dict)

gjsonpoint_dict["features"] = feat_list

# write the resulting dictionary to a .json file that outputs in the Results folder
points_path = os.path.join(outpath, 'tsp_point.json')

with open(points_path, 'w') as outfile:
    json.dump(gjsonpoint_dict, outfile, sort_keys = True, indent = 4, ensure_ascii = False)
```

Option 2: geojson module

The `geojson` module is even easier to use, and requires inputting the ordered coordinates as a list of tuples before writing the file. The code is much shorter, but is more difficult to manipulate if we need to change properties or element types within the .json file.

Use geojson module to extract coordinates and write to .json

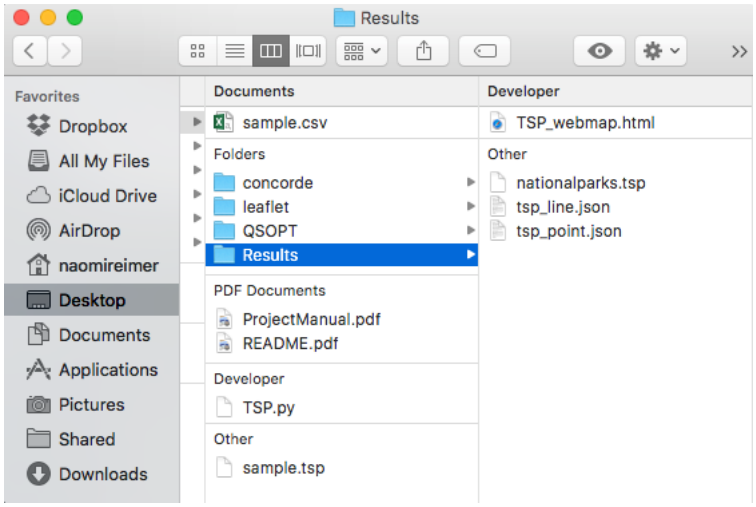
```
# extract only coordinates as list of tuples from the pandas dataframe
coordinates = []
for name in optimal_path:
    coordinates += [(dataframe.ix[name, 'Lon'], dataframe.ix[name, 'Lat'])]

# use geojson LineString method to create .json polyline file
gjsonline_dict = LineString(coordinates)

# as above, write the result to a .json file that outputs in the Results folder
line_path = os.path.join(outpath, 'tsp_line.json')
with open(line_path, 'w') as outfile:
    json.dump(gjsonline_dict, outfile, sort_keys = True, indent = 4, ensure_ascii = False)
```

Final Map Result

If we've run everything above, we should see the following file structure in the `Dailey_FinalProject/Results` folder:



If we double click on the `TSP_webmap.html`, a web map will open in our internet browser with the final result: the Euclidean distance TSP map. It should look exactly like the image below.

Congratulations! We solved the Traveling Salesman Problem!