If the math looks weird run code below and restart the notebook.

In [ ]:
```
%pip install mathjax
```

```
Requirement already satisfied: mathjax in c:\users\iansi\.conda\envs\env\lib\site-packages (0.1.2)
Note: you may need to restart the kernel to use updated packages.
```

traveling salesman problem (TSP)

Introduction

> The travelling salesman problem (TSP) seeks to the find the minimum cost hamiltonian circuit in an undirected graph, G_n.

> Or more generally, given a list of n cities, finds the fastest way to visit every city once and then return to the first city.

Problem Formulation

> Let I be a set of n cities

> Then, define an arc $C_{i,j}$ where $i, j \in I$ exists if it is possible to travel from city i to city j. And is weighted by the travel time from city i to city j.

> Next, let $X_{i,j}$ be a variable such that,

$$X_{i,j} = \begin{cases} 1 & \text{City } i \text{ connects to City } j \text{ is in the hamiltonian circuit} \\ 0 & \text{Otherwise} \end{cases}$$

> In a traditional TSP, it is possible to travel from any city to any other city, making a n complete graph, K_n. $$

This means there are $\binom{n}{2}$ total arcs.

And so there are,

$$|H(K_n)| = \frac{1}{2}(n-1)!$$

> total hamiltonian cycles in K_n $$

> This means that a brute force solution runs in exponential time, and is generally NP-Complete.

> This means that the brute force solution has poor scalability, in fact for higher values of n the solution could take years. $$

Critically; however, the problem can be cast as a network optimization problem from an adaptation of minimum spanning trees, which is potentially capable of solving the problem much faster.

Integer Programming Model Local Constraints

> Here we cover the basic formulation for a minimum tour problem.

Parameters

> Let I for a list of n cities.

> Let $C_{i,j}$ be the cost of moving from city i to city j (in travel time by road route).

Variables :

> Let $X_{i,j} \in \{0, 1\}$ be a binary variable determining if arc $C_{i,j}$ is used in the Hamiltonian circuit.

$$X_{i,j} = \begin{cases} 1 & \text{City } i \text{ connects to City } j \text{ is in the hamiltonian circuit} \\ 0 & \text{Otherwise} \end{cases}$$

> Then the goal is to find the least cost way to visit every city starting and returning to city 1.

Objective Function

> Let the cost of any circuit be,

$$\min \sum_{i \in I} \sum_{j \in I} C_{i,j} X_{i,j}$$

Local Constraints

> Subject to the constraints,

> > Entering every city only once:

$$\sum_{i \in I} X_{i,j} = 1, \qquad \forall j \in I$$

> > Leaving every city only once:

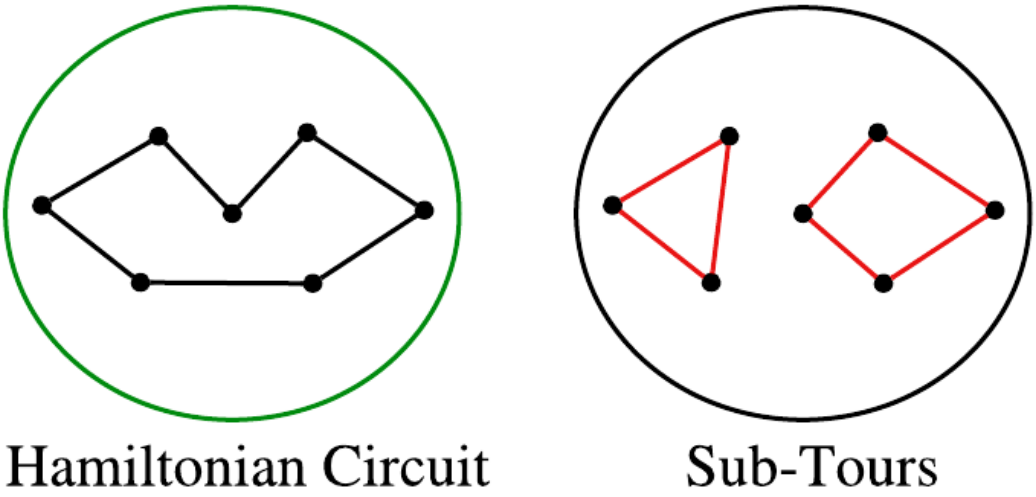
$$\sum_{k \in I} X_{j,k} = 1, \qquad \forall j \in I$$

> > Don't stay at one city:

$$X_{i,i} = 0, \qquad \forall i \in I$$

Sub Tour Problems

> Unfortunately, the local constraints above are not restrictive enough to eliminate all solutions that do not generate Hamiltonian circuits. In particular, it is still possible to get sub-tours that contribute to the final solution



> Therefore, extra constraints should be added to prevent sub-tours in the final solution.

Miller-Tucker-Zemlin 1960 (MTZ) Sub-tour Elimination

> One of the best methods to eliminate sub-tours is using the Miller-Tucker-Zemlin (MTZ) approach. Which adds a new time variable t, indicating at which point each city is visited in sequence.

New Parameter

> First fix some ordering on the list of cities I, and let $V = \{1, 2, \cdots, n\}$ be that order. So for some $i \in V$, $i = 1, 2, \cdots, n$ for n cities. $$

Then, for $i \in V$ let i denote either an integer, or a city $i \in I$.

Variable

> Next define the time variable, t_i by the time in sequence that city i is visited.

> As a result,

$$\text{if } X_{i,j} = 1, \quad \text{then} \quad t_j \geq t_i + 1, \qquad i, j \neq 1$$

> This abuses notation a bit, but in general the i, j associated with $X_{i,j}$ are city names. And, the i, j associated with t_j, t_i are integers associated with the ordering on I,

> Notice intuitively, this works by restricting a sequence on the order we visit each city. Or rather if we visit cities in the order, $i_1, i_2, \cdots, i_{n-1}$ then,

$$t_{i_1} = 1, t_{i_2} = 2, t_{i_3} = 3, \cdots, t_{i_{n-1}} = n - 1,$$

> Then, suppose a sub tour exists, such that, $\exists a, b, c \in I$ where $X_{a,b} = X_{b,c} = X_{c,a} = 1$ (cycle C_3). and none of the cities a, b, c are the originating city. ($a, b, c \neq 1$). $$

Then, the time sequence constraints impose,

$$\begin{cases} t_b \geq t_a + 1 \\ t_c \geq t_b + 1 \\ t_a \geq t_c + 1 \end{cases}$$

> Which has no solution. Similar logic can be applied to any sub-tour, (C_k), and so this method effectively eliminates sub-tours from the optimal solution.

> Lastly, consider how to encode the 'if' statement,

$$\text{if } X_{i,j} = 1, \quad \text{then} \quad t_j \geq t_i + 1, \qquad i, j \neq 1$$

> As a constraint

> In particular, consider,

$$\begin{cases} t_j \geq t_i + 1 & X_{i,j} = 1 \\ t_j \text{ Unrestricted} & X_{i,j} = 0 \end{cases}$$

> And so consider the following constraint,

$$t_j \geq t_i + 1 - M(1 - X_{i,j})$$

> For large M.

> Actually because it has been shown that there exists an ordering that ensures
>
> $$\max_{i \in V} t_i = n - 1$$

> Then selecting $M = n$ is sufficient to guarantee that an optimal Hamiltonian Circuit can satisfy the constraints.

Final Problem Formulation

Parameters

> I be a set of n cities.

> V be an ordering on I such that, $i \in I$ is either an integer i, or the $i'th$ city in the ordering of I.

> Let $C_{i,j}$ be the cost of moving from city i to city j (in travel time by road route).

Variables

> Let $X_{i,j} \in \{0, 1\}$ be a binary variable if we travel from city i to city j in the tour.
>
> $$X_{i,j} = \begin{cases} 1 & \text{City } i \text{ connects to City } j \text{ is in the hamiltonian circuit} \\ 0 & \text{Otherwise} \end{cases}$$

> Let t_i be the time city i is visited in the tour. For $i \in V$ is an integer.

Objective Function

> Let the cost of any circuit be,
>
> $$\min \sum_{i \in I} \sum_{j \in I} C_{i,j} X_{i,j}$$

Local Constraints

> > Entering every city only once:
> >
> > $$\sum_{i \in I} X_{i,j} = 1, \qquad \forall j \in I$$

> > Leaving every city only once:

$$\sum_{k \in I} X_{j,k} = 1, \qquad \forall j \in I$$

> > Don't stay at one city:

$$X_{i,i} = 0, \qquad \forall i \in I$$

MTZ Constraints

> > Sub-tour elimination

$$t_j \geq t_i + 1 - n(1 - X_{i,j}), \qquad \forall i, j \in V, \quad j > i$$

> > Circuit completeness

$$t_1 = 1$$

> The following is a file to run a TSP problem with MTZ constraint solution method. Using default parameters all results are already included in the project package. And so running any portion of the the following is optional.

Instructions

This file contains automation for running the Traveling Salesman Problem (TSP) program for user selected cities.

Note: this file uses a lot of python dependencies and API's and so is dependent on API keys and package installs. Importantly, all cells generate a file that is already included in the project folder, and so unless you want to change the default cities, running every cell is not necessary.

On a high level each section does the following:

```
>section: 'city information' >>collects which cities to use for the
TSP (you can update this list)

>section: 'Data Collection' >>converts the list of city information
to longitude latitude coordinates,
                            then calculates the travel time between
all city pairs. Making a $K_n$ complete
                            graph. (for $n$ cities).
                            Importantly, the length of each arc,
$C_{i,j}$ is the travel time from city $i$
                            to city $j$, based on the best
estimated road distance between cities.
                            And so cities that do not have feasible
driving routes will not be able to
```

```
                              be connected.
                              Data is collected from
    DistanceMatrix.ai API, https://distancematrix.ai/

    >section: 'AMPL solution' >>first generates a new .dat file using
    the distance matrix from the previous section.
                              Then, solves the TSP using Miller-
    Tucker-Zemlin (MTZ) method.
                              The result of which is printed to file
    'TSP_MTZ_TOUR_Result.txt'.

    >section: 'Display result' >>uses the file generated from the AMPL
    solution to print the optimal tour, and display
                              the tour on the US map.
```

Dependencies

> You will probably need to restart the kernal after install dependencies.

In [ ]:
```
#  Pip install dependencies....
%pip install pip
%pip install mathjax
%pip install docopt==0.6.2
%pip install geographiclib==2.0
%pip install geopy==2.3.0
%pip install matplotlib==3.6.2
%pip install numpy==1.23.3
%pip install pandas==1.5.1
%pip install plotly==5.11.0
%pip install python-dateutil==2.8.2
%pip install python-dotenv==0.21.0
%pip install requests==2.28.1
%pip install urllib3==1.26.12
%pip install amplpy
%pip install os
```

```
Requirement already satisfied: pip in c:\users\iansi\.conda\envs\env\lib\site-pack
ages (21.1.1)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: mathjax in c:\users\iansi\.conda\envs\env\lib\site-
packages (0.1.2)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: docopt==0.6.2 in c:\users\iansi\.conda\envs\env\lib
\site-packages (0.6.2)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: geographiclib==2.0 in c:\users\iansi\.conda\envs\en
v\lib\site-packages (2.0)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: geopy==2.3.0 in c:\users\iansi\.conda\envs\env\lib
\site-packages (2.3.0)Note: you may need to restart the kernel to use updated pack
ages.

Requirement already satisfied: geographiclib<3,>=1.52 in c:\users\iansi\.conda\env
s\env\lib\site-packages (from geopy==2.3.0) (2.0)
Requirement already satisfied: matplotlib==3.6.2 in c:\users\iansi\.conda\envs\env
\lib\site-packages (3.6.2)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\iansi\.conda\envs\env
\lib\site-packages (from matplotlib==3.6.2) (4.38.0)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\iansi\.conda\envs
\env\lib\site-packages (from matplotlib==3.6.2) (2.8.2)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\users\iansi\.conda\envs\env
\lib\site-packages (from matplotlib==3.6.2) (1.4.4)
Requirement already satisfied: packaging>=20.0 in c:\users\iansi\.conda\envs\env\l
ib\site-packages (from matplotlib==3.6.2) (21.3)
Requirement already satisfied: numpy>=1.19 in c:\users\iansi\.conda\envs\env\lib\s
ite-packages (from matplotlib==3.6.2) (1.23.3)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\iansi\.conda\envs\env
\lib\site-packages (from matplotlib==3.6.2) (1.0.6)
Requirement already satisfied: pillow>=6.2.0 in c:\users\iansi\.conda\envs\env\lib
\site-packages (from matplotlib==3.6.2) (9.3.0)
Requirement already satisfied: cycler>=0.10 in c:\users\iansi\.conda\envs\env\lib
\site-packages (from matplotlib==3.6.2) (0.11.0)
Requirement already satisfied: pyparsing>=2.2.1 in c:\users\iansi\.conda\envs\env
\lib\site-packages (from matplotlib==3.6.2) (3.0.9)
Requirement already satisfied: six>=1.5 in c:\users\iansi\.conda\envs\env\lib\site
-packages (from python-dateutil>=2.7->matplotlib==3.6.2) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: numpy==1.23.3 in c:\users\iansi\.conda\envs\env\lib
\site-packages (1.23.3)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: pandas==1.5.1 in c:\users\iansi\.conda\envs\env\lib
\site-packages (1.5.1)
Requirement already satisfied: python-dateutil>=2.8.1 in c:\users\iansi\.conda\env
s\env\lib\site-packages (from pandas==1.5.1) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in c:\users\iansi\.conda\envs\env\lib
\site-packages (from pandas==1.5.1) (2022.6)
Requirement already satisfied: numpy>=1.20.3 in c:\users\iansi\.conda\envs\env\lib
\site-packages (from pandas==1.5.1) (1.23.3)
Requirement already satisfied: six>=1.5 in c:\users\iansi\.conda\envs\env\lib\site
-packages (from python-dateutil>=2.8.1->pandas==1.5.1) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: plotly==5.11.0 in c:\users\iansi\.conda\envs\env\li
```

```
b\site-packages (5.11.0)Note: you may need to restart the kernel to use updated pa
ckages.

Requirement already satisfied: tenacity>=6.2.0 in c:\users\iansi\.conda\envs\env\l
ib\site-packages (from plotly==5.11.0) (8.1.0)
Requirement already satisfied: python-dateutil==2.8.2 in c:\users\iansi\.conda\env
s\env\lib\site-packages (2.8.2)Note: you may need to restart the kernel to use upd
ated packages.

Requirement already satisfied: six>=1.5 in c:\users\iansi\.conda\envs\env\lib\site
-packages (from python-dateutil==2.8.2) (1.16.0)
Requirement already satisfied: python-dotenv==0.21.0 in c:\users\iansi\.conda\envs
\env\lib\site-packages (0.21.0)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: requests==2.28.1 in c:\users\iansi\.conda\envs\env
\lib\site-packages (2.28.1)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in c:\users\iansi\.conda\envs
\env\lib\site-packages (from requests==2.28.1) (1.26.12)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\iansi\.conda\envs\en
v\lib\site-packages (from requests==2.28.1) (2022.9.24)
Requirement already satisfied: charset-normalizer<3,>=2 in c:\users\iansi\.conda\e
nvs\env\lib\site-packages (from requests==2.28.1) (2.1.1)
Requirement already satisfied: idna<4,>=2.5 in c:\users\iansi\.conda\envs\env\lib
\site-packages (from requests==2.28.1) (3.4)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: urllib3==1.26.12 in c:\users\iansi\.conda\envs\env
\lib\site-packages (1.26.12)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: amplpy in c:\users\iansi\.conda\envs\env\lib\site-p
ackages (0.8.5)
Requirement already satisfied: ampltools in c:\users\iansi\.conda\envs\env\lib\sit
e-packages (from amplpy) (0.2.8)
Requirement already satisfied: future>=0.15.0 in c:\users\iansi\.conda\envs\env\li
b\site-packages (from amplpy) (0.18.2)
Requirement already satisfied: requests==2.28.1 in c:\users\iansi\.conda\envs\env
\lib\site-packages (from ampltools->amplpy) (2.28.1)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in c:\users\iansi\.conda\envs
\env\lib\site-packages (from requests==2.28.1->ampltools->amplpy) (1.26.12)
Requirement already satisfied: charset-normalizer<3,>=2 in c:\users\iansi\.conda\e
nvs\env\lib\site-packages (from requests==2.28.1->ampltools->amplpy) (2.1.1)
Requirement already satisfied: idna<4,>=2.5 in c:\users\iansi\.conda\envs\env\lib
\site-packages (from requests==2.28.1->ampltools->amplpy) (3.4)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\iansi\.conda\envs\en
v\lib\site-packages (from requests==2.28.1->ampltools->amplpy) (2022.9.24)
Note: you may need to restart the kernel to use updated packages.
Note: you may need to restart the kernel to use updated packages.
ERROR: Could not find a version that satisfies the requirement os (from versions:
none)
ERROR: No matching distribution found for os
```

City Information

> This section selects which cities to use as inputs. Listed are the default cities,

> and so all pre-set files corespond to this list.

You can change the cities in this list and all default files will be adjusted as the notebook runs.

In [ ]:
```
from main import *

#  Add more cities if you want (The API is slow so it might take a while to collect
cities = [
    'Denver',
    'New York',
    'Houston',
    'Dallas',
    'Philadelphia',
    'Phoenix',
    'Miami',
    'Cleveland',
    'San Francisco',
    'Nashville',
    'Greensboro',
    'Lincoln',
    'Seattle'
    ]
print(cities)
```

```
['Denver', 'New York', 'Houston', 'Dallas', 'Philadelphia', 'Phoenix', 'Miami', 'C
leveland', 'San Francisco', 'Nashville', 'Greensboro', 'Lincoln', 'Seattle']
```

Data Generation

> This section prepares information from cities list.

> First by creating a list of longitude/latitude coordinates for each city.

> Then creating a distance matrix for each city pair.

(city A , city B) -> C_{A,B} = travel time from city A to city B

> Travel times are based on the optimal driving route from city A to city B.

This is collected from DistanceMatrix.ai API. https://distancematrix.ai/.

> This information is then stored in a JSON file called 'DistanceMatrix.json'

Which is read by AMPL wrapper to solve the TSP problem.

> > Importantly, sometimes the API may not be able to find a route between distant cities, in this case, that corresponding arc is neglected by getting an untractable value.

$$C_{i,j}^{neglected} = 100000 \text{ minutes}$$

> This removes the arc from any optimal tour. If it looks like this affects the optimal solution, run the API request again and it will probably find a route for the neglected arc.

In [ ]:
```
# Convert City strings to coordinate location (Takes a while)
locations = {}
for city in cities :
    locations[city] = LocationToCoordinate(city)
print(locations)
```

```
{'Denver': (39.7392364, -104.984862), 'New York': (40.7127281, -74.0060152), 'Hous
ton': (29.7589382, -95.3676974), 'Dallas': (32.7762719, -96.7968559), 'Philadelphi
a': (39.9527237, -75.1635262), 'Phoenix': (33.4484367, -112.074141), 'Miami': (25.
7741728, -80.19362), 'Cleveland': (41.4996574, -81.6936772), 'San Francisco': (37.
7790262, -122.419906), 'Nashville': (36.1622767, -86.7742984), 'Greensboro': (36.0
726355, -79.7919754), 'Lincoln': (40.8088861, -96.7077751), 'Seattle': (47.603832
1, -122.330062)}
```

In [ ]:
```
# Plots cities
import plotly.express as px
import plotly.graph_objects as go
import pandas as pd
import nbformat
df = {}
for name, coord in locations.items() :
    df[name] = {'lat' : coord[0], 'long' : coord[1], 'color' : 0.75, 'size' : 0.5,

fig = px.scatter_geo(df.values(),lat='lat',lon='long', color = 'color', size = 'siz

fig.update_geos(fitbounds="locations",scope="usa")
fig.update_layout(title = 'US map', title_x=0.5)
fig.show()
```

The following cell calculates the drive time duration between every possible city pair (i, j)

> The API is very slow, and takes about 10 minutes for 10 cities.

> If you kept the default cities, you do not need to run this again, because the results JSON, 'DistanceMatrix.json' is already included in the project folder.

> Also, this API is generated from my personal API key, which corresponds to a free account. The API key is included in a protected .env file and automatically retrieved. I don't know how well this will work after distributing the repository. So if it doesn't work.... only the default cities can be used.

In [ ]:
```
# Calculate distance between all cities  (Takes about 10 minutes for 10 cities)
```

```
# distance metric is based on the road route between two cities
# specifically is the time in minutes it takes to drive between any
# two cities. This is based on DistanceMatrix.ai open API. (Which hopefully works..
DistanceMatrix = citiesToDistanceMatrix(locations)
for city, info in DistanceMatrix.items() :
    print(city + ': ' + str([(name,travel_info['duration']) for name, travel_info i
```

Denver: [('Denver', 0.0), ('New York', 1590.65), ('Houston', 925.3166666666667), ('Dallas', 724.4833333333333), ('Philadelphia', 1555.2166666666667), ('Phoenix', 771.15), ('Miami', 1813.7666666666667), ('Cleveland', 1177.8666666666666), ('San Francisco', 1119.2833333333333), ('Nashville', 1033.9833333333333), ('Greensboro', 1439.9), ('Lincoln', 413.68333333333334), ('Seattle', 1166.9833333333333)]

New York: [('Denver', 1588.75), ('New York', 0.0), ('Houston', 1448.6666666666667), ('Dallas', 1385.7166666666667), ('Philadelphia', 104.75), ('Phoenix', 2205.0833333333335), ('Miami', 1128.9333333333334), ('Cleveland', 435.43333333333334), ('San Francisco', 2579.733333333333), ('Nashville', 804.2666666666667), ('Greensboro', 503.8666666666667), ('Lincoln', 1178.3333333333333), ('Seattle', 2532.883333333333)]

Houston: [('Denver', 935.5), ('New York', 1457.6833333333334), ('Houston', 0.0), ('Dallas', 209.23333333333332), ('Philadelphia', 1382.5166666666667), ('Phoenix', 1010.6333333333333), ('Miami', 1020.7166666666667), ('Cleveland', 1173.0166666666667), ('San Francisco', 1689.5833333333333), ('Nashville', 719.8333333333334), ('Greensboro', 989.5666666666667), ('Lincoln', 775.4166666666666), ('Seattle', 2076.5666666666666)]

Dallas: [('Denver', 721.8166666666667), ('New York', 1386.3166666666666), ('Houston', 210.41666666666666), ('Dallas', 0.0), ('Philadelphia', 1311.15), ('Phoenix', 919.85), ('Miami', 1150.45), ('Cleveland', 1042.5333333333333), ('San Francisco', 1537.5666666666666), ('Nashville', 589.35), ('Greensboro', 980.7166666666667), ('Lincoln', 570.2833333333333), ('Seattle', 1874.9166666666667)]

Philadelphia: [('Denver', 1553.3333333333333), ('New York', 110.13333333333334), ('Houston', 1365.95), ('Dallas', 1303.0166666666667), ('Philadelphia', 0.0), ('Phoenix', 2130.3166666666666), ('Miami', 1043.8666666666666), ('Cleveland', 400.0), ('San Francisco', 2544.3), ('Nashville', 721.55), ('Greensboro', 424.18333333333334), ('Lincoln', 1142.9), ('Seattle', 2497.45)]

Phoenix: [('Denver', 766.6), ('New York', 2209.25), ('Houston', 1012.95), ('Dallas', 924.5833333333334), ('Philadelphia', 2133.233333333333), ('Phoenix', 0.0), ('Miami', 2026.2666666666667), ('Cleveland', 1819.9), ('San Francisco', 689.2), ('Nashville', 1458.1833333333334), ('Greensboro', 1872.4), ('Lincoln', 1136.6166666666666), ('Seattle', 1295.95)]

Miami: [('Denver', 1813.3666666666666), ('New York', 1123.0666666666666), ('Houston', 1010.5833333333334), ('Dallas', 1143.75), ('Philadelphia', 1035.8166666666666), ('Phoenix', 2015.4833333333333), ('Miami', 0.0), ('Cleveland', 1088.1666666666667), ('San Francisco', 2678.15), ('Nashville', 782.1333333333333), ('Greensboro', 696.1333333333333), ('Lincoln', 1469.6333333333334), ('Seattle', 2904.9166666666665)]

Cleveland: [('Denver', 1179.25), ('New York', 440.5), ('Houston', 1171.3666666666666), ('Dallas', 1045.4833333333333), ('Philadelphia', 405.0), ('Phoenix', 1819.3), ('Miami', 1095.2333333333333), ('Cleveland', 0.0), ('San Francisco', 2170.2166666666667), ('Nashville', 467.26666666666665), ('Greensboro', 456.1), ('Lincoln', 768.8333333333334), ('Seattle', 2123.383333333333)]

San Francisco: [('Denver', 1126.7333333333333), ('New York', 2579.2), ('Houston', 1685.5666666666666), ('Dallas', 1540.5333333333333), ('Philadelphia', 2543.7), ('Phoenix', 686.0), ('Miami', 2685.483333333333), ('Cleveland', 2166.35), ('San Francisco', 0.0), ('Nashville', 2037.6333333333334), ('Greensboro', 2451.9), ('Lincoln', 1402.2333333333333), ('Seattle', 750.0666666666667)]

```
Nashville: [('Denver', 1058.0333333333333), ('New York', 806.0), ('Houston', 717.0
166666666667), ('Dallas', 591.1333333333333), ('Philadelphia', 730.8333333333334),
('Phoenix', 1458.0833333333333), ('Miami', 795.7333333333333), ('Cleveland', 467.2
5), ('San Francisco', 2034.2833333333333), ('Nashville', 0.0), ('Greensboro', 423.
25), ('Lincoln', 714.3), ('Seattle', 2131.4333333333334)]

Greensboro: [('Denver', 1445.3166666666666), ('New York', 498.68333333333334), ('H
ouston', 988.4666666666667), ('Dallas', 1003.55), ('Philadelphia', 417.58333333333
33), ('Phoenix', 1870.5), ('Miami', 702.5166666666667), ('Cleveland', 456.71666666
666664), ('San Francisco', 2446.7), ('Nashville', 422.0833333333333), ('Greensbor
o', 0.0), ('Lincoln', 1101.6), ('Seattle', 2480.6833333333334)]

Lincoln: [('Denver', 418.55), ('New York', 1183.45), ('Houston', 779.416666666666
6), ('Dallas', 574.5333333333333), ('Philadelphia', 1147.95), ('Phoenix', 1142.866
6666666666), ('Miami', 1474.35), ('Cleveland', 770.5333333333333), ('San Francisc
o', 1409.5166666666667), ('Nashville', 694.55), ('Greensboro', 1100.466666666666
7), ('Lincoln', 0.0), ('Seattle', 1457.2)]

Seattle: [('Denver', 1164.45), ('New York', 2538.6), ('Houston', 2077.51666666666
7), ('Dallas', 1874.7333333333333), ('Philadelphia', 2503.1), ('Phoenix', 1299.6),
('Miami', 2917.616666666667), ('Cleveland', 2125.75), ('San Francisco', 749.533333
3333333), ('Nashville', 2133.2166666666667), ('Greensboro', 2480.4333333333334),
('Lincoln', 1452.85), ('Seattle', 0.0)]
```

> Saves the distance matrix to a file, 'DistanceMatrix.json'. To be read by the AMPL solver in the next section.

In [ ]:
```
# Writes a distance matrix to a json filename.
# Later read by AMPL wrapper.
if DistanceMatrix != None :
    writeToJSON(DistanceMatrix, filename = 'DistanceMatrix.json')
#print(JsonToDict('DistanceMatrix.json'))  # Verifies that the file was created cor
```

Display Complete graph

Displays all derived arcs between the all cities.

In [ ]:
```
# Plots cities
import plotly.express as px
import plotly.graph_objects as go
import pandas as pd
import nbformat

tour_info_lat = []
tour_info_long = []
for i,city_A in enumerate(cities) :
    for city_B in cities[i:] :
        tour_info_lat += [locations[city_A][0]]
        tour_info_lat += [locations[city_B][0]]
        tour_info_lat += [None]
        tour_info_long += [locations[city_A][1]]
```

```
        tour_info_long += [locations[city_B][1]]
        tour_info_long += [None]

fig = go.Figure(data=go.Scattergeo(
    lat = tour_info_lat,
    lon = tour_info_long,
    mode = 'lines',
    line = dict(width = 2, color = 'blue'),
))

fig.update_geos(fitbounds="locations",scope="usa")
fig.update_layout(title = 'TSP all arcs on US map', title_x=0.5)
fig.show()
```

NOTE: lines between cities are shown as linear (straight segments) however, are weighted by road travel routing information. I don't have a good way to get the geometry of a route just how long it takes to travel.

AMPL Solution

> This section runs an AMPL terminal from the notebook. (It may or may not work... if it doesn't, the AMPL files will still be generated and can be run from AMPLIDE).

> Frist, AMPL_Wrapper section will make a .dat file called 'TSP_DATA.dat' from the distance matrix JSON file.

This will be used by the AMPL solution solver.

> The next section starts the AMPL terminal and runs (TSP_MTZ.mod , TSP_DATA.dat , TSP_MTZ.run).

> Importantly, in this section, the variable

>> path_to_ampl_EXE_Folder

> References the path to the folder holding AMPL.exe. Because this is just a wrapper it needs access to AMPL from local computer.

> This means to run the AMPL files from the notebook you will need to link your own path to the folder holding AMPL.exe.

>> path_to_ampl_EXE_Folder = r'Path to folder holding AMPL.exe'

> Also, the full output from the AMPL terminal will likely not load fully in the notebook.

However, it will provide an option to open the output in a new text editor.

In [ ]:
```
# Makes an AMPL .dat file from the JSON distance matrix data
from AMPL_Wrapper import *

makeDatFile(data = 'DistanceMatrix.json', outfile = 'TSP_DATA.dat')
```

Connect wrapper to AMPL.exe

Here to use AMPL on the notebook you need to link to your AMPL application folder.

And so in the variable 'path_to_ampl_EXE_Folder' place your local path to the folder containing AMPL.exe.

> path_to_ampl_EXE_Folder = r'Path to folder holding AMPL.exe'

Then run the cell.

> Or you can just run the AMPL files directly from AMPL IDE, outside of the notebook. (They will create a file called TSP_MTZ_TOUR_Results.txt that can be read by the rest of the notebook.)

In [ ]:
```
from amplpy import AMPL
from amplpy import Environment

#  The following runs ampl commands to find shortest paths. You will need to includ
#  file path to the folder containing ampl.exe, or you can just run the .dat/.mod/.
#  in the ampl ide. (This notebook just tries to automate the processes in one enti

path_to_ampl_EXE_Folder = r'C:\Users\IanSi\Downloads\amplide.mswin64\ampl.mswin64'

ampl = AMPL(Environment(path_to_ampl_EXE_Folder))

# Runs ampl files.
#  Note it is likely that the terminal result will not be fully
#  displayable in the notebook, and so it will make a new textfile
#  that contains the full result.
#  This file, TSP_MTZ.run also creates a txt file output with the
#  result (the smallest tour.) This text file is called 'TSP_MTZ_TOUR_Result.txt'
ampl.read('TSP_MTZ.mod')
ampl.read_data('TSP_DATA.dat')
ampl.read('TSP_MTZ.run')
```

```
CPLEX 20.1.0.0: mipdisplay 2
MIP Presolve modified 24 coefficients.
Reduced MIP has 170 rows, 168 columns, and 732 nonzeros.
Reduced MIP has 156 binaries, 0 generals, 0 SOSs, and 0 indicators.
Found incumbent of value 18158.800000 after 0.02 sec. (0.71 ticks)
Probing time = 0.00 sec. (0.34 ticks)
Cover probing fixed 0 vars, tightened 12 bounds.
Detecting symmetries...
MIP Presolve modified 24 coefficients.
Reduced MIP has 170 rows, 168 columns, and 732 nonzeros.
Reduced MIP has 156 binaries, 0 generals, 0 SOSs, and 0 indicators.
Probing time = 0.00 sec. (0.35 ticks)
Cover probing fixed 0 vars, tightened 12 bounds.
Clique table members: 92.
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: deterministic, using up to 8 threads.
Root relaxation solution time = 0.02 sec. (0.38 ticks)

        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer    Best Bound    ItCnt     Gap

*     0+    0                        18158.8000        0.0000           100.00%
      0     0     6547.3287    14    18158.8000     6547.3287       24   63.94%
      0     0     7444.0825    22    18158.8000      Cuts: 29       47   59.01%
      0     0     7452.3508    23    18158.8000      Cuts: 10       57   58.96%
      0     0     7459.4049    27    18158.8000      Cuts: 26       68   58.92%
      0     0     7564.9287    20    18158.8000      Cuts: 19       75   58.34%
      0     0     7693.7500    13    18158.8000      Cuts: 13       99   57.63%
Detecting symmetries...
      0     0     7704.3958    13    18158.8000      Cuts: 22      105   57.57%
      0     0     7723.5000    12    18158.8000       Cuts: 6      109   57.47%
      0     0     7734.3262    12    18158.8000   ZeroHalf: 3      117   57.41%
      0     0     7734.5067    12    18158.8000       Cuts: 8      123   57.41%
      0     0     7734.6333    12    18158.8000   ZeroHalf: 3      124   57.41%
*     0+    0                         7990.8167     7734.6333             3.21%

Clique cuts applied:  3
Cover cuts applied:  2
Mixed integer rounding cuts applied:  5
Zero-half cuts applied:  11
Gomory fractional cuts applied:  1

Root node processing (before b&c):
  Real time             =    0.14 sec. (17.18 ticks)
Parallel b&c, 8 threads:
  Real time             =    0.00 sec. (0.00 ticks)
  Sync time (average)   =    0.00 sec.
  Wait time (average)   =    0.00 sec.
                          ------------
Total (root+branch&cut) =    0.14 sec. (17.18 ticks)
CPLEX 20.1.0.0: optimal integer solution within mipgap or absmipgap; objective 799
0.816667
124 MIP simplex iterations
0 branch-and-bound nodes
absmipgap = 0.616667, relmipgap = 7.71719e-05
```

```
X [*,*]
# $2 = New_York
# $5 = Philadelphia
# $6 = Phoenix
# $8 = Cleveland
# $9 = San_Francisco
# $10 = Nashville
# $11 = Greensboro
# $12 = Lincoln
# $13 = Seattle
:             Denver  $2 Houston Dallas  $5  $6 Miami  $8  $9 $10 $11 $12 $13 :=
Denver           0     0     0      0     0   0    0    0   0   0   0   1   0
New_York         0     0     0      0     1   0    0    0   0   0   0   0   0
Houston          0     0     0      1     0   0    0    0   0   0   0   0   0
Dallas           0     0     0      0     0   1    0    0   0   0   0   0   0
Philadelphia     0     0     0      0     0   0    0    0   0   0   1   0   0
Phoenix          0     0     0      0     0   0    0    0   1   0   0   0   0
Miami            0     0     1      0     0   0    0    0   0   0   0   0   0
Cleveland        0     1     0      0     0   0    0    0   0   0   0   0   0
San_Francisco    0     0     0      0     0   0    0    0   0   0   0   0   1
Nashville        0     0     0      0     0   0    0    1   0   0   0   0   0
Greensboro       0     0     0      0     0   0    1    0   0   0   0   0   0
Lincoln          0     0     0      0     0   0    0    0   0   1   0   0   0
Seattle          1     0     0      0     0   0    0    0   0   0   0   0   0
;

t [*] :=
 1   1
 2   5
 3   9
 4  10
 5   6
 6  11
 7   8
 8   4
 9  12
10   3
11   7
12   2
13  13
;

tour_cost = 7990.82
```

Display Results

> This section displays the result from the AMPL solver.

> It will both print a list of the final tour,

> and display an image of the tour on a US map

In [ ]:
```
#  Prints the tour result from the AMPL solution

city_info = []
with open('TSP_MTZ_TOUR_Result.txt', 'r') as f:
    for line in f :
        city_info += [tuple(line.split())]
city_info.sort(key = lambda x: int(x[1]))

ordered_tour_names = []
print('TOUR')
tour_string = ''
for name,i in city_info :
    tour_string += name + ' --> '
    ordered_tour_names += [name]
ordered_tour_names += [city_info[0][0]]
tour_string += city_info[0][0]
print(tour_string)
```

```
TOUR
Denver --> Lincoln --> Nashville --> Cleveland --> New_York --> Philadelphia --> G
reensboro --> Miami --> Houston --> Dallas --> Phoenix --> San_Francisco --> Seatt
le --> Denver
```

In [ ]:
```
# Displays Optimal Tour

# Plots cities
import plotly.express as px
import plotly.graph_objects as go
import pandas as pd
import nbformat
df = {}
for name, coord in locations.items() :
    df[name] = {'lat' : coord[0], 'long' : coord[1], 'color' : 0.75, 'size' : 0.5,
tour_info = {}
for city in ordered_tour_names :
    tour_info[city.replace("_", " ")] = df[city.replace("_", " ")]

tour_info['final'] = df[ordered_tour_names[0]]


fig = px.line_geo(tour_info.values(),lat='lat',lon='long', color = 'color', hover_n

fig.update_geos(fitbounds="locations",scope="usa")
fig.update_layout(title = 'TSP MTZ solution on US map', title_x=0.5)
fig.show()
```

NOTE: lines between cities are shown as linear (straight segments) however, are weighted by road travel routing information. I don't have a good way to get the geometry of a route just how long it takes to travel.

Traditional TSP (Only using Local Constraints)

The following includes AMPL solution information for running TSP using only local constraints so solutions with sub-tours can be compared to MTZ solutions.

In [ ]:
```
from amplpy import AMPL
from amplpy import Environment

#  The following runs ampl commands to find shortest paths. You will need to includ
#  file path to the folder containing ampl.exe, or you can just run the .dat/.mod/.
#  in the ampl ide. (This notebook just tries to automate the processes in one enti

ampl = AMPL(Environment(path_to_ampl_EXE_Folder))

# Runs ampl files.
#  Note it is likely that the terminal result will not be fully
#  displayable in the notebook, and so it will make a new textfile
#  that contains the full result.
#  This file, TSP_traditional.run also creates a txt file output with the
#  result (the smallest tour.) This text file is called 'TSP_traditional_TOUR_Resul
ampl.read('TSP_traditional.mod')
ampl.read_data('TSP_DATA.dat')
ampl.read('TSP_traditional.run')
```

```
CPLEX 20.1.0.0: mipdisplay 2
Reduced MIP has 26 rows, 156 columns, and 312 nonzeros.
Reduced MIP has 156 binaries, 0 generals, 0 SOSs, and 0 indicators.
Found incumbent of value 18374.766667 after 0.02 sec. (0.48 ticks)
Probing time = 0.00 sec. (0.20 ticks)
Reduced MIP has 26 rows, 156 columns, and 312 nonzeros.
Reduced MIP has 156 binaries, 0 generals, 0 SOSs, and 0 indicators.
Probing time = 0.00 sec. (0.20 ticks)
Clique table members: 26.
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: deterministic, using up to 8 threads.
Root relaxation solution time = 0.00 sec. (0.09 ticks)

        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer    Best Bound    ItCnt     Gap

*     0+    0                        18374.7667        0.0000           100.00%
*     0+    0                        17166.0500        0.0000           100.00%
*     0+    0                        15262.2333        0.0000           100.00%
*     0+    0                        13866.0333        0.0000           100.00%
*     0+    0                        13675.6667        0.0000           100.00%
*     0+    0                        13524.9833        0.0000           100.00%
*     0     0      integral     0     6531.4167     6531.4167       11    0.00%
Elapsed time = 0.02 sec. (1.34 ticks, tree = 0.00 MB)

Root node processing (before b&c):
  Real time             =    0.02 sec. (1.34 ticks)
Parallel b&c, 8 threads:
  Real time             =    0.00 sec. (0.00 ticks)
  Sync time (average)   =    0.00 sec.
  Wait time (average)   =    0.00 sec.
                          ------------
Total (root+branch&cut) =    0.02 sec. (1.34 ticks)
CPLEX 20.1.0.0: optimal integer solution; objective 6531.416667
11 MIP simplex iterations
0 branch-and-bound nodes
X [*,*]
# $1 = Cleveland
# $4 = Greensboro
# $6 = Lincoln
# $8 = Nashville
# $9 = New_York
# $10 = Philadelphia
# $11 = Phoenix
# $12 = San_Francisco
# $13 = Seattle
:              $1 Dallas Denver  $4 Houston  $6 Miami  $8  $9 $10 $11 $12 $13 :=
Cleveland       0    0      0     0     0     0    0    1   0   0   0   0   0
Dallas          0    0      0     0     1     0    0    0   0   0   0   0   0
Denver          0    0      0     0     0     1    0    0   0   0   0   0   0
Greensboro      0    0      0     0     0     0    1    0   0   0   0   0   0
Houston         0    1      0     0     0     0    0    0   0   0   0   0   0
Lincoln         0    0      1     0     0     0    0    0   0   0   0   0   0
Miami           0    0      0     1     0     0    0    0   0   0   0   0   0
Nashville       1    0      0     0     0     0    0    0   0   0   0   0   0
```

```
New_York        0    0      0     0     0     0    0    0   0   1   0   0   0
Philadelphia    0    0      0     0     0     0    0    0   1   0   0   0   0
Phoenix         0    0      0     0     0     0    0    0   0   0   0   0   1
San_Francisco   0    0      0     0     0     0    0    0   0   0   1   0   0
Seattle         0    0      0     0     0     0    0    0   0   0   0   1   0
;

tour_cost = 6531.42
```

Collects list of tuples for city pairs

In [ ]:
```
#  Prints the tour result from the AMPL solution

city_info = []
with open('TSP_traditional_TOUR_Result.txt', 'r') as f:
    for line in f :
        city_info += [tuple(line.split())]

tour_info = {}
for city_A, city_B in city_info :
    coord_A = LocationToCoordinate(city_A.replace("_", " "))
    coord_B = LocationToCoordinate(city_B.replace("_", " "))
    tour_info[ city_A.replace("_", " ") + ' ORIG' ] = {'lat' : coord_A[0], 'long' :
    tour_info[ city_A.replace("_", " ") + ' DEST' ] = {'lat' : coord_B[0], 'long' :
    tour_info[ city_A.replace("_", " ") + ' none' ] = {'lat' : None, 'long' : None,

print('TOUR')
tour_string = ''
for city_pair in city_info :
    tour_string += str(city_pair) + '\t'
print(tour_string)
```

```
TOUR
('Denver', 'Lincoln')   ('New_York', 'Philadelphia')    ('Houston', 'Dallas')
('Dallas', 'Houston')   ('Philadelphia', 'New_York')    ('Phoenix', 'Seattle')
('Miami', 'Greensboro') ('Cleveland', 'Nashville')      ('San_Francisco', 'Phoeni
x')     ('Nashville', 'Cleveland')      ('Greensboro', 'Miami') ('Lincoln', 'Denve
r')     ('Seattle', 'San_Francisco')
```

Displays Tour solution on US Map.

In [ ]:
```
# Displays Optimal Tour

# Plots cities
import plotly.express as px
import plotly.graph_objects as go
import pandas as pd
import nbformat

fig = go.Figure(data=go.Scattergeo(
    lat = [info['lat'] for info in tour_info.values()],
    lon = [info['long'] for info in tour_info.values()],
    mode = 'lines',
    line = dict(width = 2, color = 'blue')
))
```

```
fig.update_geos(fitbounds="locations",scope="usa")
fig.update_layout(title = 'TSP local constraints solution on US map', title_x=0.5)
fig.show()
```

NOTE: lines between cities are shown as linear (straight segments) however, are weighted by road travel routing information. I don't have a good way to get the geometry of a route just how long it takes to travel.

The above display shows the result of the local constraint TSP. Notice, the result is likely to have sub-tours.