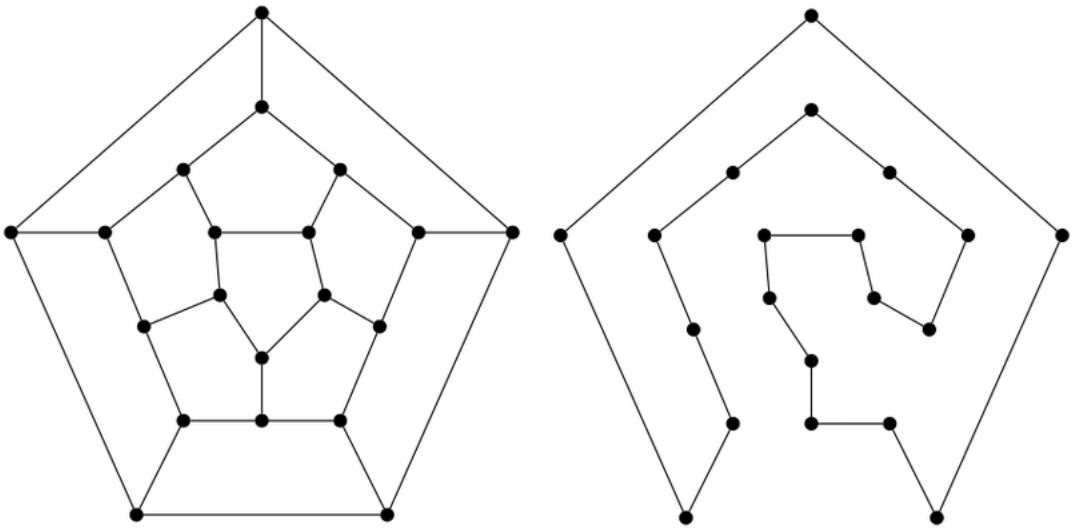
# Discrete II Final Project

MATH 307 — Spring 2019

**Garrett Brenner**

April 28, 2019

# Abstract




For my Discrete II Final Project, I have elected to focus on a real-world application of *The Travelling Salesman Problem*. My goal is to use a graphing algorithm to find the shortest path between each state in the US and to model this with an animation. I use data collected from the Bureau of Transportation Statistics to model flight connections between each of these states (excluding Hawaii and Alaska, as they are not located within the same continent).

# Mathematical Theory

The Travelling Salesman Problem could be briefly summarized as such: given a set of cities and the distances between them (if such a path exists), find the shortest possible route that visits every city in the set exactly once and then returns to the starting city.

In Graph Theory, a Hamiltonian Cycle exists on a graph $G = (V, E)$ if there exists a path between each vertex $v \in V$ that only touches each vertex once and returns to the starting vertex, in essence, creating a cycle.

If each city is considered a vertex and each route between them an edge, it can be seen that the TSP resembles the problem of finding the shortest possible Hamiltonian Cycle for $G =$ (the set of cities , the set of paths between cities).

This problem is NP-hard as well as NP-complete, essentially stating that an algorithm's best case completion time is the same as its worst case and furthermore that its worst case increases as much as exponentially with each additional city. This factor comes into play later on.

# Procedure

I began by searching for a list of flight information that I could use to build a graph. I found a .csv exportable list of regularly scheduled flights on the Bureau of Transportation Statistics website. The unaltered file had 4123 rows as well as several additional columns which I deleted until all that remained was "Row Number", "Origin", and "Destination".

| 1 | December 2018: Regularly Scheduled Flights Canceled 5% or More | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | Airline: All airlines | | | | | | | |
| 3 | | | | | | | | |
| 4 | Row Numb | Airline | Flight No. | Origin | Destinatior | Total Oper | Number Ca | Percent_Cancelled |
| 5 | 1 | AA | 327 | ATL | CLT | 1 | 1 | 100 |
| 6 | 2 | AS | 1953 | SAN | SFO | 28 | 28 | 100 |
| 7 | 3 | C5 | 4990 | RIC | EWR | 2 | 2 | 100 |
| 8 | 4 | EV | 4084 | EWR | BNA | 1 | 1 | 100 |
| 9 | 5 | EV | 3978 | LAN | ORD | 1 | 1 | 100 |
| 10 | 6 | EV | 2869 | GRI | DFW | 1 | 1 | 100 |

To be a true Travelling Salesman problem designed to visit each state only once, I only needed one airport from each state to represent visiting the state as a whole. To accomplish this, I went through the list of states in the US, excluding Hawaii and Alaska and selected what I assumed to be the most popular (sharing the most edges with other airports) airport for each one and adding their airport codes to a list. The reason for choosing the most popular airports was to aid increasing the probability of a multitude of Hamiltonian Cycles between them.

*Set of Airports*: {MGM, ANC, PHX, LIT, SMF, DEN, BDL, ILG, MCO, ATL, HNL, BOI, ORD, IND, DSN, MCI, SDF, MSY, AUG, BWI, BOS, DTW, MSP, JAN, STL, BIL, LNK, LAS, MHT, EWR, ABQ, JFK, CLT, BIS, CMH, OKC, PDX, PHL, PVD, CHS, PIR, BNA, DFW, SLC, MPV, RIC, SEA, CRW, MSN, CYS}

At this point, not having a desire to manually edit a 4123 row long Excel sheet, I knew I required the help of someone more experienced with data science. I tried applying what I knew of the pandas an numpy libraries in Python to the data and Googling my way through, but to no avail. Thankfully, Data Science and Economics double major Mackenzie Schaich was willing to give up a few minutes of her time to help me. All work was done in Jupyter Lab through Anaconda.

The first thing we needed to do was alphebatize the "Origin" column to make the spreadsheet more readable and make it easier to identify errors.

```
[59]: df = df.sort_values('Origin')
```

```
[60]: df.head()
```

[60]:

| 2 | Origin | Destination |
|---|---|---|
| **1964** | ABE | CLT |
| **3371** | ABE | CLT |
| **2542** | ABI | DFW |
| **1126** | ABQ | OAK |
| **1062** | ABQ | PHX |

Following this, all we had to do was remove all airports that were not in my list of one airport per state which was a .txt file with each airport delimited by a new line.

[61]:
```
airports_list = pd.read_csv('listofairports.txt', sep='\n')
```

[62]:
```
airports_list = list(airports_list['MGM'])
```

[63]:
```
for i in range(len(airports_list)):
    airports_list[i] = airports_list[i][:3]
```

[75]:
```
df = df.loc[(df.Origin.isin(airports_list)) & (df.Destination.isin(airports_list))]
```

[77]:
```
df = df.drop_duplicates()
```

[78]:
```
df.head()
```

[78]:

| | 2 | Origin | Destination |
|---|---|---|---|
| **1062** | | ABQ | PHX |
| **1016** | | ABQ | LAS |
| **1004** | | ABQ | BWI |
| **3587** | | ABQ | ORD |
| **3595** | | ABQ | PDX |

Now that I had the data split into alphebatized chunks where I could easily observe all destination airports from each origin airport, I added a third column titled "Distance". Finally, I manually computed each distance using airmilescalculator.com and entered them into Excel. Next it was time to convert the .csv to .txt and open it up in Python.

At this point I was also considering how I was going to be graphing these airports, hopefully in a way that loosely resembled their location on a map of the United States. In retrospect, it would have been a better idea to just get the longitudinal and latitudinal coordinates for each of them, but at the time I hadn't thought of that. For each airport in my list of included airports, I manually looked up and wrote down a working address and then did a bulk entry into mapcustomizer.com to view them all on a map of the US. After that, I was planning on using Photoshop to find the pixel distance between the center of the model and each individual airport in order to hopefully provide a basis for graphical representation, but that plan didn't pan out.

I spent several days trying to install the iGraph library for Python to help with the visualization, but I was never able to do it successfully. Here, I feel I must interject my personal distaste for that library and its monster of an installment process. Even now, it attempts to lure me with its beautiful visualizations, but I shall not give in for I know it is unachievable.

After that flopped, I was forced to do the entire project with no visualization which was unfortunate, but did make it easier to code. I found an applicable algorithm on geeksforgeeks.org written in C++ that solved the minimum Hamiltonian cycle of a graph given an adjacency matrix and a source vertex and then returned the total weight of that path. It was only a matter of translating that C++ into Python and making a few tweaks here and there before I was able to get it working on a sample graph provided by the website. At this point, I took the liberty of also implementing the ability to view the entire Hamiltonian path, not just the total weight. In layman's terms, whereas before the screen might show 8000 miles to represent the minimum distance visiting each state and returning, it now would also show the sequence of airports one would take to achieve such a path.

```
'''this is the Traveling Salesman Problem algorithm transcribed from the
geeksforgeeks weblink you sent me. I have tested it and it works.
To run it, you must pass in an nxn integer matrix and integer "source"'''
def tsp(graph, src):
    vertex = []
    for i in range(len(graph[0])):
        if i != src:
            vertex.append(i)

    min_pathweight = sys.maxsize
    min_path = [sys.maxsize]

    #do start
    while True:
        current_pathweight = 0
        current_path = []

        k = src
        for i in range(len(vertex)):
            current_pathweight += graph[k][vertex[i]]
            current_path.append([k,vertex[i]])
            k = vertex[i]
        current_pathweight += graph[k][src]
        current_path.append([k,src])
        current_path.append(current_pathweight)

        min_pathweight = min(min_pathweight, current_pathweight)
        check1 = int(min_path[len(min_path)-1])
        check2 = int(current_path[len(current_path)-1])

        minimum = min(check1, check2)
        if minimum == check2:
            min_path = current_path

    #do end
        #while start
        if not next_permutation(vertex):
            break
        #while end

    if min_pathweight >= sys.maxsize:
        return "no path"
    else:
        #return min_pathweight
        return min_path
```

Here's where the time complexity comes into play. As was noted by the author of the geeksforgeeks article, the time complexity of the algorithm was $O(n!)$ where $n$ is the number of cities. For example, calculating the minimum cycle with four cities takes four times as long as calculating it with three cities because

$$(4)! = 4 \cdot (3)! = 4 \cdot 3 \cdot (2)! = 4 \cdot 3 \cdot 2 \cdot 1$$

My data contained 42 unique airports for the algorithm to work with.

$$(42)! \approx 1400000000000000000000000000000000000000000000000$$

So you can try to comprehend how long that might take for a computer to solve. This meant I would not be able to solve the original problem I had set out to solve, not because I couldn't find a solution, but simply because it would just take too long. However, solving smaller problems like with 10 airports instead of 42 is easy for my computer. Below demonstrates an adjacency matrix (e.g. the distance going from $\text{airport}_0$ to $\text{airport}_2$ is represented by $M_{0,2}$) for 10 airports: DFW, ABQ, PHX, LAS, BWI, ORD, PDX, DEN, MCI, MCO, ATL. If an element in the matrix is $\infty$ that means there is no path between those two airports (at least in that direction, as $M_{0,2}$ may not exist but $M_{2,0}$ can).

$$\begin{bmatrix} 0 & 569 & 868 & 1055 & 1217 & 802 & 1616 & 641 & 460 & 985 \\ 569 & 0 & 328 & 486 & 1670 & 1118 & 1111 & 349 & 718 & 1553 \\ 868 & 328 & 0 & \infty & \infty & 1440 & 1009 & 602 & 1044 & \infty \\ 1055 & 486 & 255 & 0 & \infty & \infty & \infty & \infty & \infty & \infty \\ 1217 & 1670 & 1999 & \infty & 0 & \infty & \infty & \infty & \infty & 787 \\ 802 & 1118 & 1440 & \infty & \infty & 0 & \infty & \infty & 403 & 1005 \\ 1616 & \infty & 1009 & \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ 641 & 349 & 602 & 628 & 1491 & \infty & \infty & 0 & 533 & \infty \\ 460 & \infty & 1044 & \infty & \infty & 403 & \infty & \infty & 0 & \infty \\ 985 & 1553 & \infty & \infty & 787 & \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

Professor Kumar connected me with the College of Charleston's high performance computing so I altered my code to begin at a 4x4 adjacency matrix and work its way up to a 42 x 42 adjacency matrix and sent it over just to see how far it could get. I also changed the source vertex to be the Dallas Fort Worth airport because it has the most connections.

# Results and Conclusion

The highest result I was able to get on my computer within a reasonable amount of time was using an 11x11 matrix. The returned path and path weight were as follows:

[['DFW', 'LAS'], ['LAS', 'ABQ'], ['ABQ', 'PDX'], ['PDX', 'PHX'], ['PHX', 'DEN'], ['DEN', 'MCI'], ['MCI', 'ORD'], ['ORD', 'ATL'], ['ATL', 'BWI'], ['BWI', 'MCO'], ['MCO', 'DFW'],]
8154 miles is the minimum distance

The highest result I was able to get on the high performance computer within a reasonable amount of time was using a blankxblank matrix. The returned path and weight were as follows:

[] m miles is the minimum distance

# References

Singh, N. (2017, November 11). Traveling Salesman Problem (TSP) Implementation. Retrieved from https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/