Modeling Chicago as a Weighted Congestion Network

James Conlon

5/4/2017

UIC UNIVERSITY OF ILLINOIS AT CHICAGO
Department of Civil and Materials Engineering
COLLEGE OF ENGINEERING

Abstract

The purpose of this project was to model Chicago as a network, given traffic congestion values. Chicago's congestion network is designed to represent the city in real time since data utilized is as up to date as possible. Concepts of network science are straightforward, provided that a simulation or scripting tool is utilized. Applications in this paper are straightforward and require little mathematical or programming knowledge. Geographic visualizations paired with simple figures make this concept easy to understand. This paper starts with assumptions about Chicago as a whole, then goes though methodology process of creating a network model of Chicago. Since congestion values can easily be translated into units of travel time, the final portion of this paper dedicates itself to explain how network science can solve routing problems in transportation.

A Brief Introduction

Traffic congestion is a problem facing urban areas. It results in adverse environmental impact as a result of greenhouse gas emission from fuel combustion in vehicles. Time losses also arise from congestion. This paper examines how travel times are effected by congestion, as modeled by a network. The requirements of a network are straightforward and simple. It is a graph consisting of nodes (points) and links, which connect nodes. Networks can be applied nearly anything, but in this case networks are used to model a transportation system. Congestion, as used in this paper refers to a vehicle's anticipated travel speed through a specific portion of street (also referred to as a segment). The units are the same as speed, given in miles per hour.

Data

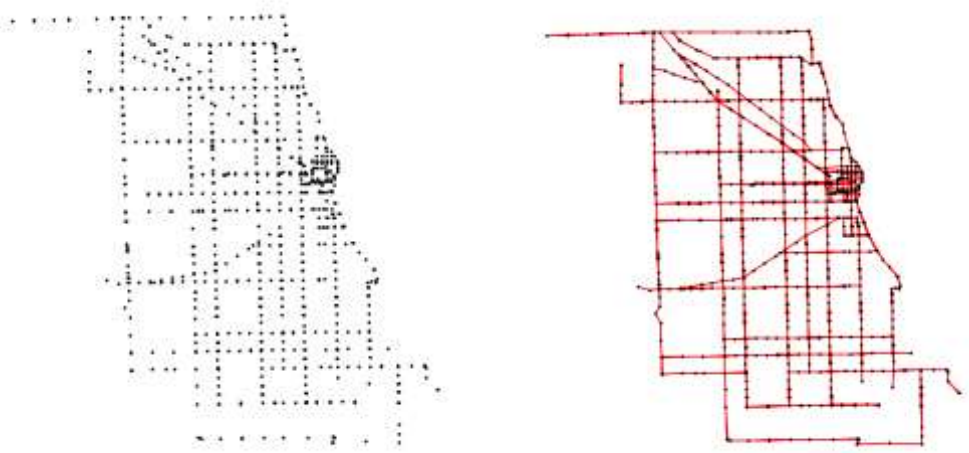
The data utilized in this project is available publically through the City of Chicago Data Portal. Traffic congestion values were obtained through the *Congestion Estimates by Segment* database. Congestion estimates are available for 1,257 street segments in Chicago, and are updated as frequently as every ten minutes. These congestion values are given as a speed (miles per hour) and are estimated with the help of the Chicago Transit Authority (CTA) Bus Tracker system. Data is collected from buses along their routes and traffic speed estimates are calculated. The length (in miles) of the segment, along with its start and end coordinates is also provided.

It is unreasonable to assume that traffic congestion data is readily available for the entirety of Chicago's street network. Therefore, the first general assumption made in this project was that Chicago's streets can be simplified and represented as the street segments with available

congestion estimates as provided in the data portal. Below is a comparison of Chicago's complete streets and Chicago's streets as represented by the network.

Data sets can be readily downloaded from *data.cityofchicago.org* as a .csv file, but data collection for this project's analysis was done through Python queries for automation and efficiency purposes.

Start and end locations of the given segments do not align perfectly to specific locations or intersections in Chicago. The coordinates of the start and end points are provided, and with enough significant digits to accurately plot. In this network, a node was defined as a latitude and longitude pair represented as either the start or end coordinate of a segment specified in the data portal. To avoid rounding error, floating point issues, and creating extraneous nodes, coordinates were rounded to four decimal places. For reference, 0.0001° latitude and 0.0001° longitude are roughly equivalent to 11 meters and 8 meters, respectively. In total, there are 799 nodes in this network. Shown below is a plot of nodes and node connections (street segments), performed through the Python library *NetworkX*.



The plot looks almost identical to the previous figure, which was generated through GIS software. The street network plot does not exactly overlay actual streets in Chicago, but demonstrates connections between nodes. Since the length of each congestion segment is provided through the data portal, it is not necessary to use a GIS to calculate geometric features.

At this point, nodes and links have been established. In network science, links can be assigned varying values. This can be used as a *weight* for each connection. An *unweighted* network ignores any attributes that links hold. Links are only used to indicate connection between nodes. In a *weighted* network, links hold value. This can be thought of as a *cost* function between connected nodes. Congestion speed is the main variable provided in this analysis, given in miles per hour. This is not a suitable variable for a weighted matrix since it does not represent a cost quantity, but rather a ratio. Travel time along links is more suitable for this analysis. This represents a cost (time required to travel between nodes).

The travel time between segments can be easily be calculated as the segment's given length divided by travel speed (congestion value) through the segment.

$$travel\ time = \frac{length}{congestion}$$

Performing unit analysis to get a unit of seconds:

$$travel\ time[\text{s}] = length[\text{mi}] \times (\frac{1}{congestion})[\frac{\text{h}}{\text{mi}}] \times 60[\frac{\text{min}}{\text{hour}}] \times 60[\frac{\text{s}}{\text{min}}]$$

The above equation is utilized in this project's Python analysis script to calculate travel time estimates for street segments.

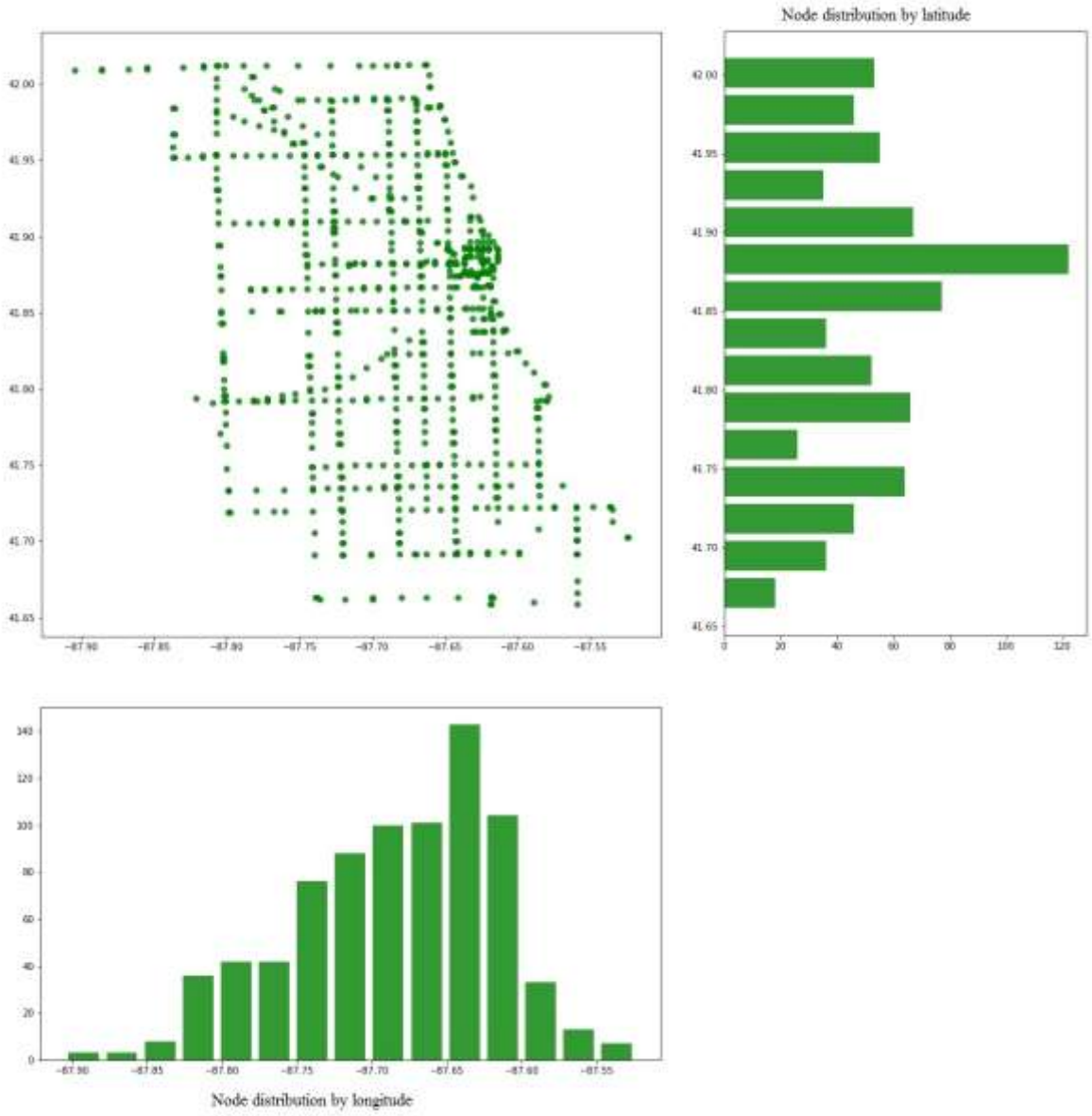
It should be noted that not all segments are given new updates when the data portal refreshes. Also, if no information is available for the data portal to calculate a congestion estimate, a dummy value of -1 is given for that segment. Using the negative dummy values in the travel time calculation would cause inaccuracy, so some assumptions were made and implemented in Python. A data collection script was ran constantly and updated a file containing the latest values. If a value of -1 was retrieved when calculating travel time, the assumption was that the congestion value was that of what was last kept in the file containing the last values. If there was no record in that file (e.g. the value was -1 again), the current average of the latest congestion data pull was utilized.

The reasoning behind the congestion value assumptions was that the dummy values did not create an inaccessible node pair. For example, if there was no data available for a segment (read connection between two nodes), it did not indicate that the edge between the node pair was removed. A vehicle could still travel between the nodes. This assumption does result in some inaccuracies, but prevents isolated networks from appearing.

A network can also be classified by its preference to direction. In a *directed* network, links are dependent on starting and ending node. For example, node one may have a connection link to node two, but node two may not necessarily have a connection link to node one. In *Undirected* networks, a link indicates a connection both ways. Transportation networks share characteristics to both types. One street can possibly be traveled along in both directions like an undirected network. A one-way street would be an example of a directed transportation network. Considering a weighted network adds complexity to this, since congestion values may differ between directions. All that being said, this project's Chicago network was assumed to be an undirected network. The shortcoming with this assumption is that weight (time) values may be representative of the opposite direction. This assumption does reduce the chance of missing data and isolated sub networks.

Creating a network with thousands of pieces is not difficult with Python. Here, the library *NetworkX* is your friend. The Chicago congestion network can be created by passing reference to node locations, and congestion segments that connect nodes. A more detailed explanation of Python's role in network analysis is provided later. For now, assume that Chicago has been properly set up with Python.

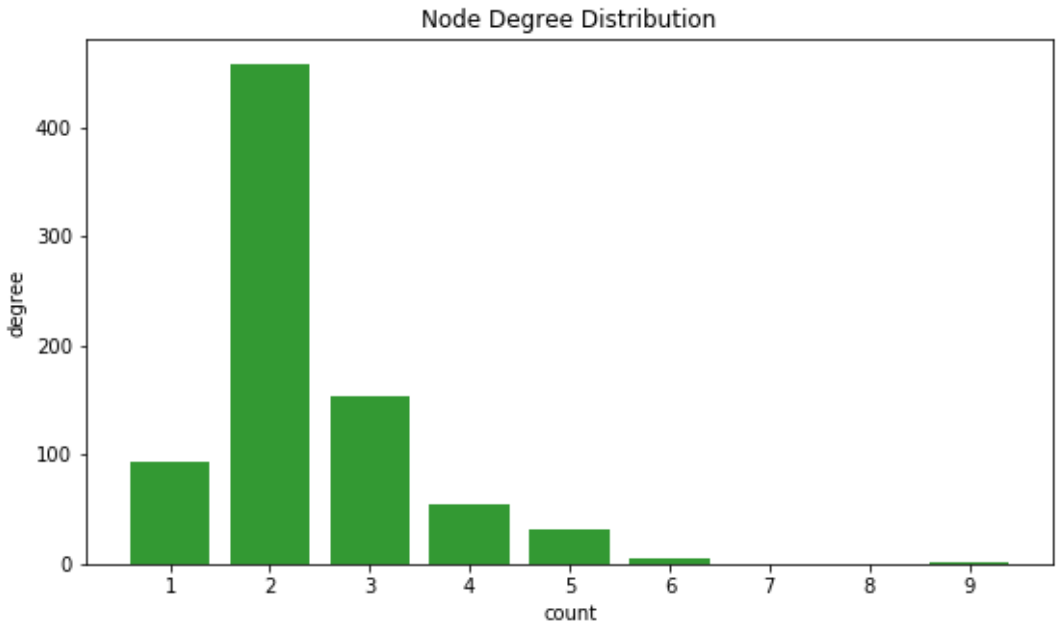
Now that a network has been set up, let's perform some analysis on Chicago's streets. Since the network nodes are set up by their geographical coordinates, a spatial distribution histogram can be created with respect to latitude and longitude



Not every node is directly connected by a link. In a *complete* network, a link exists between all possible node pairs. Real world examples are never complete networks. Transportation and city networks are *sparse* and have far fewer connections than a complete network would contain. In this example network of Chicago, there are only around 0.2% of the total number of possible links (the number of links if the network was complete). A complete transportation network is

completely unrealistic in a complex city. This would require an individual street connecting all locations to each other in a single segment.

A node's *degree* is simply the count of connections to a separate node. Shown below is a distribution of the 799 nodes by their corresponding degree.



This plot is consistent with large networks. That is, there is a large number of nodes with low degrees (e.g. one or two connections) and a small number of nodes with higher degrees. A degree of two is more common than one here since the network we are looking at is transportation related. A node with a degree of one would signify either a dead end or a geographic bound of the spatial area covered by the network. The end result roughly follows a lognormal distribution, peaking at degree two.

As a whole, the degree of connectivity can be calculated as

$$\gamma_p = \frac{L}{3N - 6} = 0.4$$

The subscript *p* indicates that the network is planar, as this transportation network represents streets accessible to one another.

A few more network metrics can be calculated for Chicago's streets. A *Betti* number is simply the number of links divided by the number of nodes and can represent complexity.

$$B = \frac{L}{N} = 1.2$$

The number of links is higher than the number of nodes, indicating complexity in Chicago. In application, this means that in general, there are multiple transit routes that exist connecting nodes. Since this network was assumed to be undirected, the average number of connected nodes would be twice this value, or 2.4. This is because two connections exist for each link.

Recall that this network is not completely representative of Chicago since not all streets can be taken into consideration. These network metrics can only be taken as an estimate.

The *CongestionNetwork* Library

A Python library named *CongestionNetwork* was written for this analysis. Essentially, it is a wrapper around *NetworkX* that uses the Chicago Data Portal information to create a real time congestion network of Chicago. Detail for this library is available in the appendix.

Chicago can be initialized as a network object. Creating this object pulls data and assigns nodes, links, and current congestion values for Chicago. Recall earlier that data may result in dummy

values. For this analysis, a data collection script was continuously ran to fill in missing values. This is also included in the appendix.

After the network is created, a few methods can be called. plotNetwork plots the network in its entirety, as a weighted network with travel times color coded from a scale of green to red. That is, lower travel time is represented as green and color changes through yellow, orange, and red, as travel time increases.

Below is an example of two plotted networks, around thirty minutes apart.

A .pdf reference file is also provided. This shows in more detail, the network node indexes and the connecting paths. Since it is not an image, the document can be searched. This means that the location of a specified node can be quickly found.

The library also provides a method to plot specific plots. An index of nodes is passed, and the path is mapped. This is useful later when shortest paths are discussed. Below is an example of *plotPath*.

Application in Routing

Chicago has now been modeled as a network, weighted by travel time estimates. The travel time for a trip from one node to another is simply the sum of the weighted edges that connect the nodes. In a small network, this task is easy. Chicago streets represent a more complex network. There are multiple path options between nodes, and the desired path can depend on many factors. Someone driving to work may want to minimize travel time from their home node to their work node. A tourist may wish to find the path with the most scenery. A manufacturer can try to determine the route that minimizes fuel cost for their truck fleet.

Theoretically, there exists an optimum path between two nodes on a weighted network, provided they are not isolated from one another. The goal would be to find the path that minimizes a specific cost function. As explained before, the cost for this weighted network would be travel time.

Imagine trying to compute manually which path to take between two nodes in a complex network. One technique may be to find every possible path and calculate the time cost to reach the desired node. This would require much computational power and would be inefficient. Another technique may be to select only a few and calculate time cost, selecting the best given option. This is quick but may leave out more efficient paths.

Computer scientists and mathematicians have developed algorithms to address this problem. *NetworkX* in Python implements these algorithms. For a weighted network, a *Dijkstra* path can be computed. The computation behind the algorithm is not as important as its use, since it is abstracted into only a few lines of code when implemented in Python.

The code:

```
shortest_path = nx.dijkstra_path(g,242,400)
```

Returns a list of the computed shortest path between node 242 and node 546. These two points are arbitrary without a corresponding location. Node 242 is the furthest northwest node in the network, and node 400 is on the northern coast of the city.

Plotting the shortest path is not directly passible as a function in *NetworkX*. For the purpose of this project, a library called *CongestionNetwork* was created.

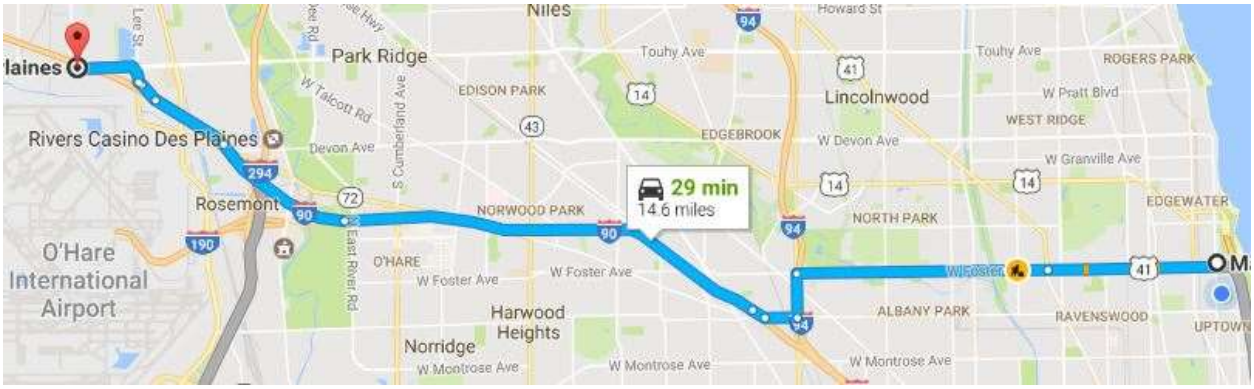
Running:

```
Chicago = CongestionNetwork.Network(query=query)
g = Chicago.plotNetwork()

path_graph = Chicago.plotPath(shortest_path)
```

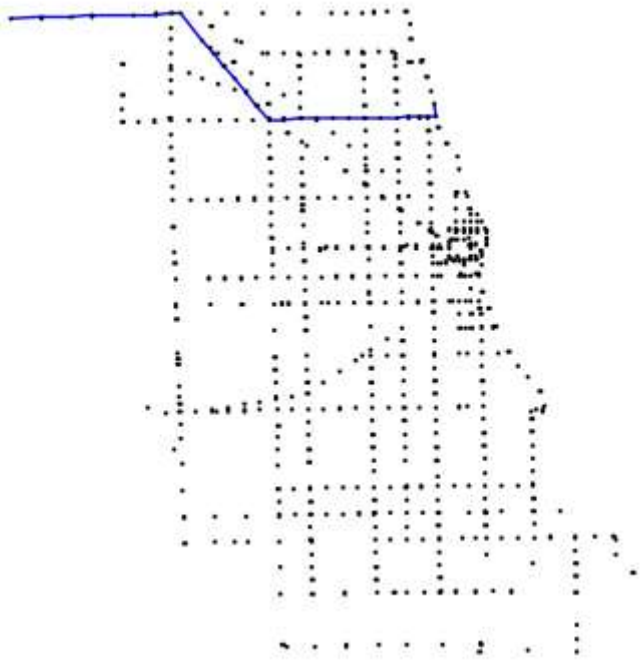
Results in

The length of the shortest path in this case was calculated to be 2716 seconds, or roughly 45 minutes. This needs to be compared to a commonly used method like Google Maps. The node coordinates of 242 are geographically equivalent to where I-90 passes over Touhy Ave, just North of O'hare Airport. Node 400 corresponds to the intersection of Foster Ave and Sheridan on the Edgewater-Uptown border. Below is the result of Google Map directions between the locations.

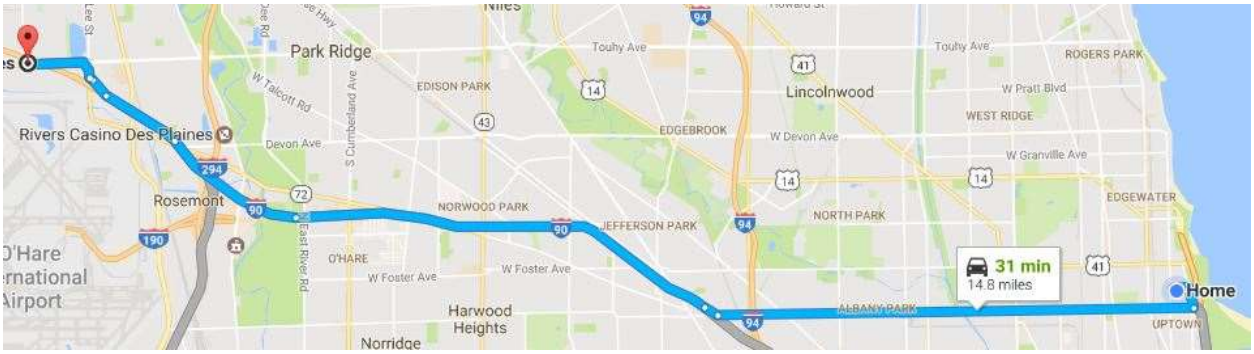


Google Maps calculated the fastest travel time as 29 minutes. The error from this congestion network approach was 55%, but the traveled path was similar.

For sensitivity analysis purposes, a different ending node was selected. This node (numbered 64) was a few blocks south of the previous end node. When the network model was ran, the following results were obtained.



Shortest path was calculated to be 2717 seconds, or 45 minutes. Notice that the selected path was different for this run. For a similar Google Maps inquiry:



A result of 31 minutes was obtained. The model was off by 14 minutes.

Routing Conclusions

The routes selected using the congestion network were comparable to a similar route chosen by Google Maps. Convenience of a service like Google Maps makes the congestion model currently undesirable for route selection. A few features should be included in the next version of this model.

1. Congestion estimates are also provided by traffic region from the Chicago Data Portal. If these regions are mapped, missing values can be filled in by the value of the segment's corresponding region. Theoretically, this would provide a more accurate congestion value than pulling older data or estimating the entire network average.

2. Another shortcoming is that nodes are not geolocated in a convenient way. Using latitude and longitude coordinates makes for easy plotting, but is not an intuitive way for users to add input. These should ideally be intersections, landmarks, buildings, or other recognizable points in Chicago.

3. This initial model represents an undirected network, as explained previously. Congestion values are pulled as directional. If directions are added, the model would be more accurate, but would eliminate some routing options.

If these key problems are addressed, the next model may yield comparable results to a service like Google Maps.

Appendix A CongestionNetwork Library

```
class Network:
    import numpy as np
    import pandas as pd
    import matplotlib.pyplot as plt



    def __init__(self, query,last_value_file_string='last_values.csv'):

        import pandas as pd
        import numpy as np

        self.query = query
        self.last_value_file_string = last_value_file_string
        self.last_values_pd  = pd.read_csv(last_value_file_string)
        self.last_values_congestion = self.last_values_pd['congestion'].values


        usecols = ['segmentid','street','_fromst','_tost','_length','_traffic','start_lon','_lif_lat','_lit_lon','_lit_lat']
        raw_data = pd.read_csv(query,dtype='str',usecols=usecols)
        length = len(raw_data)

        start_lon = raw_data['start_lon'].values.astype(np.float)
        start_lat = raw_data['_lif_lat'].values.astype(np.float)
        end_lon = raw_data['_lit_lon'].values.astype(np.float)
        end_lat  = raw_data['_lit_lat'].values.astype(np.float)
        self.seg_ids = raw_data['segmentid'].values.astype('str')
        seg_lengths = raw_data['_length'].values.astype(np.float)
        self.congestion = raw_data['_traffic'].values.astype('int')

        def getLastValue(idx,val='index'):
            if(val=='segid'):
                index = np.where(self.seg_ids == str(idx))[0][0]
            else:
                index = idx
            cong_value = self.last_values_congestion[index]
            return(cong_value)

        start_coor_array = []
        end_coor_array  = []
        self.node_stack = []
        for i in range(length):
           seg_id_temp = self.seg_ids[i]
           start_lon_temp = "{:.3f}".format((start_lon[i]))
           start_lat_temp = "{:.3f}".format((start_lat[i]))
           end_lon_temp = "{:.3f}".format((end_lon[i]))
           end_lat_temp = "{:.3f}".format((end_lat[i]))
           start_coor_temp = (start_lon_temp,start_lat_temp)
           start_coor_array.append(start_coor_temp)
           end_coor_temp = (end_lon_temp,end_lat_temp)
           end_coor_array.append(end_coor_temp)
           self.node_stack.append([seg_id_temp,start_lon_temp,start_lat_temp,end_lon_temp,end_lat_temp])

        self.node_stack = np.array(self.node_stack)
        self.all_coor = start_coor_array
        self.all_coor.extend(end_coor_array)
        self.unique = list(set(self.all_coor))

        self.unique_array = np.array(self.unique)
        unique_lon = self.unique_array[:,0]
        unique_lat = self.unique_array[:,1]
        self.connections = []
        self.node_labels = []
        self.missing_idx = []
        self.weight_values = []
        self.congestion_speed_values = []
        self.connections_array = []

        for i in range(length):
            self.node_labels.append(i)
            from_lon, from_lat  = start_coor_array[i]
            to_lon, to_lat = end_coor_array[i]
```

```
        from_node = np.where((unique_lon== from_lon) & (unique_lat == from_lat))[0]
        to_node = np.where((unique_lon== to_lon) & (unique_lat == to_lat))[0]

        congestion_temp = self.congestion[i]
        length_temp = seg_lengths[i]
        if ((congestion_temp != -1) and (congestion_temp != 0)):
            weight = (1/congestion_temp)*length_temp*60*60
            self.weight_values.append(weight)
            self.congestion_speed_values.append(congestion_temp)
        else:
            self.missing_idx.append(i)
            last_value = getLastValue(i)
            if (last_value != -1):
                weight = last_value
            else:
                #weight = 1000 #too high to be considered a path
                #weight =
                if(len(self.congestion_speed_values)>0):
                    avg_cong = sum(self.congestion_speed_values)/len(self.congestion_speed_values)
                else:
                    avg_cong = 15
                weight = (1/avg_cong)*length_temp*60*60

        self.connections.append((from_node[0],to_node[0],weight))
        self.connections_array.append([from_node[0],to_node[0],weight])
    self.connections_array = np.array(self.connections_array)
```

```
def plotNetwork(self,cmap=plt.cm.RdYlGn_r,node_color='k',node_size=50,size_x=18,size_y=18,save=False,fname='network_file.png'): #YlGn_r
    import numpy as np
    import matplotlib.pyplot as plt
    import networkx as nx


    scale_min = round(min(self.weight_values))-1
    scale_max = round(max(self.weight_values))-1

    g = nx.Graph()
    for i, coordinate in enumerate(np.asarray(self.unique,float)):
        g.add_node(i, pos = coordinate, label = self.node_labels[i])

    pos= nx.get_node_attributes(g,'pos')

    g.add_weighted_edges_from(self.connections)
    edges,weights = zip(*nx.get_edge_attributes(g,'weight').items())
    self.weight_array = np.array(weights)

    plt.figure(1,figsize=(size_x,size_y))
    nx.draw(g,pos,node_size = node_size, node_color=node_color,edge_color=weights,edgelist=edges,width=5.0, edge_cmap=cmap)#,v_min=scale_min,vmax=scale_max)
    #nx.draw(g,pos,with_labels=True,node_size = 10, font_color='k',font_size=4, node_color='y',edge_color='b',edgelist=edges,width=2.0)#, edge_cmap=cmap)
    #plt.savefig('reference_map.pdf')
    if(save==True):
        plt.savefig(fname)

    plt.show()

    return(g)

def plotPath(self,path,node_color='k',edge_color='b',node_size=50):
    import numpy as np
    import matplotlib.pyplot as plt
    import networkx as nx

    g = nx.Graph()
    for i, coordinate in enumerate(np.asarray(self.unique,float)):
        g.add_node(i, pos = coordinate, label = self.node_labels[i])

    pos= nx.get_node_attributes(g,'pos')

    path_len = len(path)
    arr = []
    for n in range(path_len-1):
        temp_ = (path[n],path[n+1])
        arr.append(temp_)
    g.add_edges_from(arr)
    plt.figure(1,figsize=(18,18))
    nx.draw(g,pos,node_size=node_size,node_color=node_color,edge_color=edge_color,width=5)
    plt.show()
    return(g)
```

Appendix B Data Collection

```
import pandas as pd
import numpy as np
import time
import CongestionNetwork

query='https://data.cityofchicago.org/resource/8v9j-bter.csv?%24limit=5000&%24%24app_token=1oBcClovBSmSkEGPHggrWWW6X'


backup_file = 'last_values_backup.csv'
backup_data = pd.read_csv(backup_file)
start_over = False #change to true to overwrite data

total_len = 1257
if(start_over == True):
    last_values = [-1]*total_len
else:
    last_values = backup_data['congestion'].values

test = False
hour = 0
while(True):
    print(hour,time.localtime())
    #hour= hour+=1
    current_day = time.localtime()[2]
    current_hour = time.localtime()[3]
    current_minute = time.localtime()[4]
    query='https://data.cityofchicago.org/resource/8v9j-bter.csv?%24limit=5000&%24%24app_token=1oBcClovBSmSkEGPHggrWWW6X'
    usecols = ['segmentid','_traffic']
    raw_data = pd.read_csv(query,dtype='str',usecols=usecols)
    cong_values = raw_data['_traffic'].values
    seg_ids = raw_data['segmentid'].values
    missing_idx = np.where(cong_values==-1)[0]

    segment_values = raw_data['segmentid'].values
    missing_segs = segment_values[missing_idx]

    temp_cong_array = []
    for i in range(total_len):
        if(cong_values[i] != -1):
            temp_cong_array.append(cong_values[i])
            last_values[i] = cong_values[i]
        else:
            temp_cong_array.append(last_values[i])

    print('missing count:',temp_cong_array.count(str(-1)))
    print('available:',(total_len -temp_cong_array.count(str(-1))))


    Chicago = CongestionNetwork.Network(query=query)
    img_fname = 'img_{0}_{1}_{2}.png'.format(current_day,current_hour,current_minute)
    g = Chicago.plotNetwork(save=True,fname=img_fname)

    #time_series_file_name = 'output_{0}_{1}_{2}.csv'.format(current_day,current_hour,current_minute)
    file_name = 'last_values.csv'
    output_stack = np.column_stack((seg_ids,temp_cong_array))
    output_df = pd.DataFrame(output_stack,columns=['segment','congestion'])
    out_csv = output_df.to_csv(file_name)
    time.sleep(900) #10 minutes
```

References

Data:

Chicago Data Portal (*data.cityofchicago.org*)

Python:

Libraries: *Numpy*, *Pandas*, *Matplotlib*, *NetworkX*, *CongestionNetwork*

Note: CongestionNetwork was made specifically for this project

Routing:

Google Maps, Google Earth