Lab 03 Graph Theory

October 8, 2019

In this lab I convert a GRF format to other representations of a graph, use a graph to find paths between nodes, and to solve a traveling salesman problem.

1 Graph Representation

```
In [17]: import numpy as np

         def myFunc(file, graph_selection):

             """
             myFunc is the main function for part one. It acts as the driver
             function to read in a grf format file and outputs a desired
             converted graph. Within the conditional statements a seperate function
             is called that provides the logic for deired outputs.

             Input:
                 file(string) - file name of grf format file
                 graph_selection(int) - int selection between 1 to 4 for
                 desired converted graph
             Output:
                 G(str) - graph to be displayed
             """

             #open file
             f = open(file)
             #read the lines of the fule
             str_data = f.readlines()
             #for each element in the file, split the string and convert to float
             data = [[float(element) for element in str.split()] for str in str_data]

             #edge list
             if graph_selection == 1:
                 print('Edge List for {}'.format(file))
                 G = edgeList(data)

             #Adjacency matrix
```

```
        if graph_selection == 2:
            #print('Adjacency Matrix for {}'.format(file))
            G = adjacencyMatrix(data)

        #Adjacency Structure
        if graph_selection == 3:
            print('Adjacency Structure for {}'.format(file))
            G = adjacencyStructure(data)

        #incidence Matrix
        if graph_selection == 4:
            print('Incidence Matrix for {}'.format(file))
            G = incidenceMatrix(data, edgeList(data))

        #return graph
        return G
```

In [2]:
```
def edgeList(data):

    """
    edgeList converts nodes and its connected nodes into a list format.
    This function reads in data and and outputs the formatted edge list.

    Input:
        data(list) - list of lists that provide info about a graph
    Output:
        edge_list(str) - str formatted as edge list
    """

    #create empty edge list
    edge_list = []

    #loop through grf data
    for i in data:
        #loop through edges in formated grf file
        for j in i[3:]:
            #assign the edge to the node
            edge = [i[0],j]
            #if first node is less than paired node
            #this removes duplicates
            if (edge[0] < edge[1]):
                #add node and its corresponding node
                #that shares an edge
                edge_list.append(edge)

    #return edge list
    return edge_list
```

In [3]:
```
def adjacencyMatrix(data):
```

```
    """
    adjacencyMatrix converts nodes to lists of edges to find an adjacency
    matrix that provides the information on connected nodes. Connected
    nodes are represented by 1, 0 otherwise.

    Input:
        data(list) - list of lists that provide info about a graph
    Output:
        adj_matrix(array) - sparce adjacency matrix
    """

    #grab all nodes from grf file
    nodes = [sub_list[0] for sub_list in data]
    #set var n to the length of nodes
    n = len(nodes)

    #initialize empty lists to hold pairs of edges
    edge_1= []
    edge_2 = []

    #loop through data
    for i in data:
        #loop through paired edges
        for j in i[3:]:
            #assign edge to node
            mat = [i[0],j]
            #append edges
            edge_1.append(mat[0])
            edge_2.append(mat[1])

    #create empty adjacency matrix with zeros
    adj_matrix = np.zeros((n,n))

    #loop through edge list
    for i in range(len(edge_1)):
        #unpack list and convert to int to be able to index
        u = int(edge_1[i])
        v = int(edge_2[i])
        #assign 1.0 to connected nodes
        adj_matrix[u-1][v-1] = 1.0

    #return adjacency matrix
    return adj_matrix
```

In [4]:
```
def adjacencyStructure(data):

    """
```

```
        adjacencyStructure converts nodes to a list of nodes and adjacent nodes.
        the nodes and adjacent nodes are then zipped together and converted to
        a dictionary to provide keys for the connected nodes.

        Input:
            data(list) - list of lists that provide information of a graph
        Output:
            adj_struc(dist) - dictionary of nodes and and connected nodes as keys
        """

        #create lists of nodes and adjacencent nodes
        nodes = [sub_list[0] for sub_list in data]
        adj_nodes = [sub_list[3:] for sub_list in data]

        #zip up lists and convert to dict to form structure
        adj_struct = dict(zip(nodes, adj_nodes))

        #output graph
        return adj_struct
```

```
In [5]: def incidenceMatrix(data, edge_list):

        """
        incidenceMatrix converts nodes to a list of nodes and uses the edges_list
        to create matrix

        Input:
            data(list) - list of lists that provide information of a graph
            edge_list(str) - str formatted as an edge list
        Output:
            inc_matrix(array) - sparce incidence matrix
        """

        #get all nodes from grf format
        nodes = [sub_list[0] for sub_list in data]
        #get the length of all nodes for size of matrix
        n = len(nodes)

        #initialize incidence matrix by how many edges there are and nodes
        #4x5
        inc_matrix = np.zeros((len(edge_list), n))

        for i in range(len(edge_list)):
            for j in range(len(edge_list[i])):
                inc_matrix[i, int(edge_list[i][j] - 1)] = 1
                inc_matrix[i, int(edge_list[i][j] - 1)] = 1

        #output graph
```

```
            return inc_matrix
```

1.1 Converting GRF format

```
In [6]: #Edge list
        myFunc('graph.grf', 1)

Edge List for graph.grf


Out[6]: [[1.0, 2.0], [1.0, 3.0], [2.0, 3.0], [3.0, 4.0]]

In [7]: #Adjacency matrix
        myFunc('graph.grf', 2)

Adjacency Matrix for graph.grf


Out[7]: array([[0., 1., 1., 0., 0.],
               [1., 0., 1., 0., 0.],
               [1., 1., 0., 1., 0.],
               [0., 0., 1., 0., 0.],
               [0., 0., 0., 0., 0.]])

In [8]: #Adjacency structure
        myFunc('graph.grf', 3)

Adjacency Structure for graph.grf


Out[8]: {1.0: [2.0, 3.0], 2.0: [1.0, 3.0], 3.0: [1.0, 2.0, 4.0], 4.0: [3.0], 5.0: []}

In [9]: #incidence matrix
        myFunc('graph.grf', 4)

Incidence Matrix for graph.grf


Out[9]: array([[1., 1., 0., 0., 0.],
               [1., 0., 1., 0., 0.],
               [0., 1., 1., 0., 0.],
               [0., 0., 1., 1., 0.]])
```

1.2 Converting GRF format of South America

```
In [24]: import pandas as pd

         #display nodes and country dataframe
         data = {"Country":["Argentina", "Bolivia", "Brazil", "Chile",
                            "Columbia", "Ecuador", "Guyana", "Paraguay",
```

```
                          "Peru", "Suriname", "Uruguay", "Venezuela"]}
df = pd.DataFrame(data)
df.columns.name="Node"
df.index += 1
print("Countries are organized in alphabetical order")
df
```

```
Countries are organized in alphabetical order
```

Out[24]:

Node	Country
1	Argentina
2	Bolivia
3	Brazil
4	Chile
5	Columbia
6	Ecuador
7	Guyana
8	Paraguay
9	Peru
10	Suriname
11	Uruguay
12	Venezuela

In [11]:
```
#Edge list
myFunc('southAmerica.grf', 1)
```

```
Edge List for southAmerica.grf
```

Out[11]:
```
[[1.0, 2.0],
 [1.0, 3.0],
 [1.0, 4.0],
 [1.0, 8.0],
 [1.0, 11.0],
 [2.0, 3.0],
 [2.0, 4.0],
 [2.0, 8.0],
 [2.0, 9.0],
 [3.0, 5.0],
 [3.0, 7.0],
 [3.0, 8.0],
 [3.0, 9.0],
 [3.0, 10.0],
 [3.0, 11.0],
 [3.0, 12.0],
 [4.0, 9.0],
 [5.0, 6.0],
 [5.0, 9.0],
```

```
          [5.0, 12.0],
          [6.0, 9.0],
          [7.0, 10.0],
          [7.0, 12.0]]
```

```
In [12]: #Adjacency matrix
         myFunc('southAmerica.grf', 2)
```

```
Adjacency Matrix for southAmerica.grf
```

```
Out[12]: array([[0., 1., 1., 1., 0., 0., 0., 1., 0., 0., 1., 0.],
                [1., 0., 1., 1., 0., 0., 0., 1., 1., 0., 0., 0.],
                [1., 1., 0., 0., 1., 0., 1., 1., 1., 1., 1., 1.],
                [1., 1., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
                [0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1.],
                [0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0.],
                [0., 0., 1., 0., 0., 0., 0., 0., 0., 1., 0., 1.],
                [1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0.],
                [0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
                [1., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 1., 0., 1., 0., 1., 0., 0., 0., 0., 0.]])
```

```
In [13]: #Adjacency Structure
         myFunc('southAmerica.grf', 3)
```

```
Adjacency Structure for southAmerica.grf
```

```
Out[13]: {1.0: [2.0, 3.0, 4.0, 8.0, 11.0],
          2.0: [1.0, 3.0, 4.0, 8.0, 9.0],
          3.0: [1.0, 2.0, 5.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0],
          4.0: [1.0, 2.0, 9.0],
          5.0: [3.0, 6.0, 9.0, 12.0],
          6.0: [5.0, 9.0],
          7.0: [3.0, 10.0, 12.0],
          8.0: [1.0, 2.0, 3.0],
          9.0: [2.0, 3.0, 4.0, 5.0, 6.0],
          10.0: [3.0, 7.0],
          11.0: [1.0, 3.0],
          12.0: [3.0, 5.0, 7.0]}
```

```
In [14]: #incidence matrix
         myFunc('southAmerica.grf', 4)
```

```
Incidence Matrix for southAmerica.grf
```

```
Out[14]: array([[1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [1., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [1., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
                [1., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
                [1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
                [0., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 1., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
                [0., 0., 1., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
                [0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
                [0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
                [0., 0., 1., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
                [0., 0., 1., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
                [0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
                [0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
                [0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 0.],
                [0., 0., 0., 0., 1., 1., 0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0.],
                [0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 1.],
                [0., 0., 0., 0., 0., 1., 0., 0., 1., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0., 1., 0., 0., 1., 0., 0.],
                [0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 1.]])
```

1.3 Counting Walks

We can use an adjacency matrix to count walks. The *ij*'th entry of $A^k v$ will give the number of k-length paths connecting to nodes *i and j*.

```
In [25]: #Count the number of walks from Chile to Venezuela

         #create the adjacency matrix for the South America grf graph
         A = myFunc('southAmerica.grf', 2)

         #vector v is the starting point at Chile
         #in this case Chile is the fourth country in the grf file
         #thus I create a vector v to be zero except for a 1 in the
         #entry corresponging to node B (Chile)
         v = [0,0,0,1,0,0,0,0,0,0,0,0]

         #to get the fewest number of countries that I must visit,
         #including Chile and Venezuela I raise A to the power of 3
         #looking at a map the min num of countries is 4 (including c and v)
         #power of 2 doesnt work since there are zero terms in the list
         #which indicate that the graph isnt connected
         walk_c2v = np.dot(np.linalg.matrix_power(A, 3), np.transpose(v))[-1]

         #all shortest walks from Chile to Venezuela
```

```
all_walks = [int(i) for i in np.dot(np.linalg.matrix_power(A, 3), np.transpose(v))]

print('Fewest number of countries that must be visited, including Chile and Venezuela
print('\nAll shortest walks from Chile to Venezuela')
print('1. Chile, Peru, Columbia, Venezuela')
print('2. Chile, Peru, Brazil, Venezuela')
print('3. Chile, Bolivia, Brazil, Venezuela')
print('4. Chile, Argentina, Brazil, Venezuela')
```

```
Fewest number of countries that must be visited, including Chile and Venezuela is 4.
There are 4 possible paths.

All shortest walks from Chile to Venezuela
1. Chile, Peru, Columbia, Venezuela
2. Chile, Peru, Brazil, Venezuela
3. Chile, Bolivia, Brazil, Venezuela
4. Chile, Argentina, Brazil, Venezuela
```

```
In [20]: num_walks_8c = np.dot(np.linalg.matrix_power(A, 8), np.transpose(v)).sum()
         print("The total amount of different possible walks of size 8, from Chile is {:,} wal
```

```
The total amount of different possible walks of size 8, from Chile is 137,225 walks
```

We can generalize the method used above and remove dependence on vector v. Let A be the adjacency matrix from a graph G. Then $A^k_{i,j}$ counts the number of walks of length k from node *i to node j*. To find the total number of walks of size 8, just sum the total number of walks.

```
In [21]: #different walks of size 8
         num_walks8 = np.linalg.matrix_power(A, 8).sum()
         print('The total amount of different possible walks of size 8 is {:,} walks'.format(i
```

```
The total amount of different possible walks of size 8 is 1,909,772 walks
```

2 Traveling Salesman Problem

```
In [26]: #imort external modules
         import itertools
         import numpy as np

         def tsp(filename):

             #read the text file using list comprehension
             data = [[float(x) for x in str.split()] for str in open(filename).readlines()]

             #read the edge list of the txt file
```

```
edges = [x[0:2] for x in data]

#read in the edge weights of the txt file
weights = [x[2] for x in data]

#create permutations
city_num = [2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]
perms = list(itertools.permutations(city_num))
perms = [list(x) for x in perms]

#number of nodes
n = 8

#initialize empty lists to hold pairs of edges
edge_1= []
edge_2 = []

#loop through data
for i in edges:
    edge_1.append(i[0])
    edge_2.append(i[1])

#create empty edge matrix with infinity
edge_matrix = np.full((n,n), np.inf)
#set diagonal equal to zero
np.fill_diagonal(edge_matrix, 0)

#loop through edge list
for i in range(len(edge_1)):
    #unpack list and convert to int to be able to index
    u = int(edge_1[i])
    v = int(edge_2[i])
    #assign weights to connected nodes
    edge_matrix[u-1][v-1] = weights[i]
    edge_matrix[v-1][u-1] = weights[i]

#initialize distance array 1 x 5040
dist = np.zeros(len(perms))
permutation = []

#for loop that loops through original set of permutations
#i also creates an index that is used to insert distance into
#the distance array
for i,perm in enumerate(perms):
    #add a 1 to the beginning of the permutation list
    perm.insert(0,1.0)
    #add a 1 to the end of the permutation list
    perm.append(1.0)
```

```
            permutation.append(perm)
            #get indexes through the lengh of the num of permutations -1
            for j in range(len(perm)-1):
                #find the distance of the permutations
                #index at each coordinate of the edge matrix using the perm array
                dist[i] = dist[i] + edge_matrix[int(perm[j])-1, int(perm[j+1])-1]

        #optimal city list and reversed list
        optimal_city = permutation[np.argmin(dist)]

        #display city order and list
        print("Since we are starting and ending at the same city there are two optimal tra
        print(optimal_city)
        print("Atlanta, Charlotte, Jacksonville, Miami, Tallahassee, Mobile, Montgomery, N
        print(optimal_city[::-1])
        print("Atlanta, Nashville, Montgomery, Mobile, Tallahassee, Miami, Jacksonville, C


        #get the minimum travel distance
        min_dist = np.min(dist)

        #output minimum distance
        print("The minimum travel time for the salesman is {} hours".format(min_dist))

        return min_dist
```

```
In [27]: min_dist = tsp('salesman.txt')
```

```
Since we are starting and ending at the same city there are two optimal travel paths
 with the same travel time. The two paths are the same, just in revered order.

[1.0, 2.0, 3.0, 4.0, 8.0, 5.0, 6.0, 7.0, 1.0]
Atlanta, Charlotte, Jacksonville, Miami, Tallahassee, Mobile, Montgomery, Nashville, Atlanta

[1.0, 7.0, 6.0, 5.0, 8.0, 4.0, 3.0, 2.0, 1.0]
Atlanta, Nashville, Montgomery, Mobile, Tallahassee, Miami, Jacksonville, Charlotte, Atlanta

The minimum travel time for the salesman is 35.25 hours
```