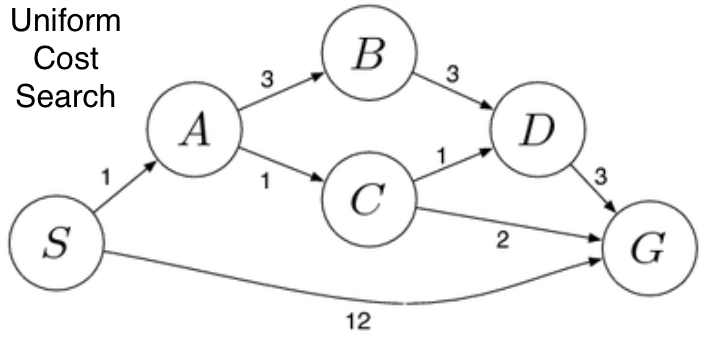
**LAB # 7**

#### Uniform Cost Search

This algorithm is mainly used when the step costs are not the same but we need the optimal solution to the goal state. In such cases, we use Uniform Cost Search to find the goal and the path including the cumulative cost to expand each node from the root node to the goal node. It does not go depth or breadth, it searches for the next node with the lowest cost and in the case of the same path cost, let’s consider lexicographical order in our case.



In the above figure consider S to be the start node and G to be the goal state. From node S we look for a node to expand and we have nodes A and G but since it’s a uniform cost search it’s expanding the node with the lowest step cost so node A becomes the successor rather than our required goal node G. From A we look at its children nodes B and C. So since C has the lowest step cost it traverses through node C and then we look at successors of C i.e. D and G since the cost to D is low we expand along with the node D. Since D has only one child G which is our required goal state we finally reach the goal state D by implementing UFS Algorithm. If we have traversed this way definitely our total path cost from S to G is just 6 even after traversing through many nodes rather than going to G directly where the cost is 12 and 6<<12(in terms of step cost). But this may not work with all cases.

## **Analysis on a Dataset**

We will be looking to take a generic dataset (not one that is specifically intended to be used for Graphs) and do some manipulation (in pandas) so that it can be ingested into a Graph in the form of a edge list. And edge list is a list of tuples that contain the vertices defining every edge

The dataset we will be looking at comes from the Airlines Industry. It has some basic information on the Airline routes. There is a Source of a journey and a destination. There are also a few columns indicating arrival and departure times for each journey. As you can imagine this dataset lends itself beautifully to be analysed as a Graph. Imagine a few cities (nodes) connected by airline routes (edges). If you are an airline carrier, you can then proceed to ask a few questions like

What is the shortest way to get from A to B? In terms of distance and in terms of time

Is there a way to go from C to D?

Which airports have the heaviest traffic?

Which airport in “in between” most other airports? So that it can be converted into a local hub

import pandas as pd

import numpy as np

data = pd.read\_csv('data/Airlines.csv')

data.shape

(100, 16)

data.dtypes

YEAR int64

MONTH int64

DAY int64

DAY\_OF\_WEEK int64

AIRLINE object

FLIGHT\_NUMBER int64

TAIL\_NUMBER object

ORIGIN\_AIRPORT object

DESTINATION\_AIRPORT object

SCHEDULED\_DEPARTURE int64

DEPARTURE\_TIME float64

DEPARTURE\_DELAY float64

TAXI\_OUT float64

SCHEDULED\_TIME int64

ELAPSED\_TIME float64

AIR\_TIME float64

DISTANCE int64

TAXI\_IN float64

SCHEDULED\_ARRIVAL int64

ARRIVAL\_TIME float64

ARRIVAL\_DELAY float64

DIVERTED int64

dtype: object

1. We notice that origin and destination look like good choices for Nodes. Everything can then be imagined as either node or edge attributes. A single edge can be thought of as a journey. And such a journey will have various times, a flight number, an airplane tail number etc associated with it
2. We notice that the year, month, day and time information is spread over many columns. We want to create one datetime column containing all of this information. We also need to keep scheduled and actual time of arrival and departure separate. So we should finally have 4 datetime columns (Scheduled and actual times of arrival and departure)
3. Additionally, the time columns are not in a proper format. 4:30 pm is represented as 1630 instead of 16:30. There is no delimiter to split that column. One approach is to use pandas string methods and regular expressions
4. We should also note that sched\_dep\_time and sched\_arr\_time are int64 dtype and dep\_time and arr\_time are float64 dtype
5. An additional complication is NaN values

# converting sched\_dep\_time to 'std' - Scheduled time of departure

data['std'] = data.SCHEDULED\_DEPARTURE.astype(str).str.replace('(\d{2}$)', '') + ':' + data.SCHEDULED\_DEPARTURE.astype(str).str.extract('(\d{2}$)', expand=False) + ':00'

# converting sched\_arr\_time to 'sta' - Scheduled time of arrival

data['sta'] = data.SCHEDULED\_ARRIVAL.astype(str).str.replace('(\d{2}$)', '') + ':' + data. SCHEDULED\_ARRIVAL.astype(str).str.extract('(\d{2}$)', expand=False) + ':00'

# converting dep\_time to 'atd' - Actual time of departure

data['atd'] = data.DEPARTURE\_TIME.fillna(0).astype(np.int64).astype(str).str.replace('(\d{2}$)', '') + ':' + data. DEPARTURE\_TIME.fillna(0).astype(np.int64).astype(str).str.extract('(\d{2}$)', expand=False) + ':00'

# converting arr\_time to 'ata' - Actual time of arrival

data['ata'] = data.ARRIVAL\_TIME.fillna(0).astype(np.int64).astype(str).str.replace('(\d{2}$)', '') + ':' + data. ARRIVAL\_TIME.fillna(0).astype(np.int64).astype(str).str.extract('(\d{2}$)', expand=False) + ':00'

We now have time columns in the format we wanted. Finally we may want to combine the year, month and day columns into a date column. This is not an absolutely necessary step. But we can easily obtain the year, month and day (and other) information once it is converted into datetime format.

data['date'] = pd.to\_datetime(data[['YEAR', 'MONTH', 'DAY']])

# finally we drop the columns we don't need

data = data.drop(columns = ['YEAR', 'MONTH', 'DAY'])

Now import the dataset using the networkx function that ingests a pandas dataframe directly. Just like Graph creation there are multiple ways Data can be ingested into a Graph from multiple formats.

**import** networkx **as** nx

FG = nx.from\_pandas\_edgelist(data, source='ORIGIN\_AIRPORT', target='DESTINATION\_AIRPORT', edge\_attr=True,)

FG.nodes()

Output:

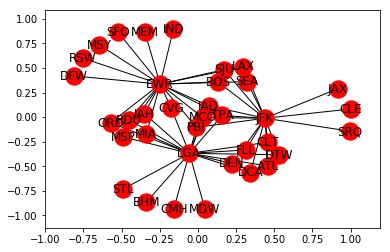
NodeView(('EWR', 'MEM', 'LGA', 'FLL', 'SEA', 'JFK', 'DEN', 'ORD', 'MIA', 'PBI', 'MCO', 'CMH', 'MSP', 'IAD', 'CLT', 'TPA', 'DCA', 'SJU', 'ATL', 'BHM', 'SRQ', 'MSY', 'DTW', 'LAX', 'JAX', 'RDU', 'MDW', 'DFW', 'IAH', 'SFO', 'STL', 'CVG', 'IND', 'RSW', 'BOS', 'CLE'))

FG.edges()

Output:

EdgeView([('EWR', 'MEM'), ('EWR', 'SEA'), ('EWR', 'MIA'), ('EWR', 'ORD'), ('EWR', 'MSP'), ('EWR', 'TPA'), ('EWR', 'MSY'), ('EWR', 'DFW'), ('EWR', 'IAH'), ('EWR', 'SFO'), ('EWR', 'CVG'), ('EWR', 'IND'), ('EWR', 'RDU'), ('EWR', 'IAD'), ('EWR', 'RSW'), ('EWR', 'BOS'), ('EWR', 'PBI'), ('EWR', 'LAX'), ('EWR', 'MCO'), ('EWR', 'SJU'), ('LGA', 'FLL'), ('LGA', 'ORD'), ('LGA', 'PBI'), ('LGA', 'CMH'), ('LGA', 'IAD'), ('LGA', 'CLT'), ('LGA', 'MIA'), ('LGA', 'DCA'), ('LGA', 'BHM'), ('LGA', 'RDU'), ('LGA', 'ATL'), ('LGA', 'TPA'), ('LGA', 'MDW'), ('LGA', 'DEN'), ('LGA', 'MSP'), ('LGA', 'DTW'), ('LGA', 'STL'), ('LGA', 'MCO'), ('LGA', 'CVG'), ('LGA', 'IAH'), ('FLL', 'JFK'), ('SEA', 'JFK'), ('JFK', 'DEN'), ('JFK', 'MCO'), ('JFK', 'TPA'), ('JFK', 'SJU'), ('JFK', 'ATL'), ('JFK', 'SRQ'), ('JFK', 'DCA'), ('JFK', 'DTW'), ('JFK', 'LAX'), ('JFK', 'JAX'), ('JFK', 'CLT'), ('JFK', 'PBI'), ('JFK', 'CLE'), ('JFK', 'IAD'), ('JFK', 'BOS')])

nx.draw\_networkx(FG, with\_labels=True) # Quick view of the Graph. As expected we see 3 very busy airports



nx.algorithms.degree\_centrality(FG) # Notice the 3 airports from which all of our 100 rows of data originates

nx.density(FG) # Average edge density of the Graphs

Output:

0.09047619047619047

nx.average\_degree\_connectivity(FG) # For a node of degree k - What is the average of its neighbours' degree?

Output:

{1: 19.307692307692307, 2: 19.0625, 3: 19.0, 17: 2.0588235294117645, 20: 1.95}

As is obvious from looking at the Graph visualization (way above) – There are multiple paths from some airports to others. Let us say we want to calculate the shortest possible route between 2 such airports. Right off the bat we can think of a couple of ways of doing it

* There is the shortest path by distance
* There is the shortest path by flight time

What we can do is to calculate the shortest path algorithm by weighing the paths with either the distance or airtime. Please note that this is an approximate solution – The actual problem to solve is to calculate the shortest path factoring in the availability of a flight when you reach your transfer airport + wait time for the transfer. This is a more complete approach and this is how humans normally plan their travel. For the purposes of this article we will just assume that is flight is readily available when you reach an airport and calculate the shortest path using the airtime as the weight

Let us take the example of JAX and DFW airports:

# Let us find all the paths available

for path in nx.all\_simple\_paths(FG, source='JAX', target='DFW'):

print(path)

# Let us find the dijkstra path from JAX to DFW.

dijpath = nx.dijkstra\_path(FG, source='JAX', target='DFW')

dijpath

Output:

['JAX', 'JFK', 'SEA', 'EWR', 'DFW']

# Let us try to find the dijkstra path weighted by airtime (approximate case)

shortpath = nx.dijkstra\_path(FG, source='JAX', target='DFW', weight=' AIR\_TIME ')

shortpath

Output:

['JAX', 'JFK', 'BOS', 'EWR', 'DFW']

##### **Lab Tasks**

# **Task 1:**

# Find the shortest path from SEA to EWR weighted by distance.

# **Task 2:**

# Find the shortest path from JFK to BOS weighted by airtime.

# **Task 3:**

# What are the top 10 airlines based on the number of flights with the help of visualization?

# **Task 4:**

# What are the top 10 departure airports? Visualize by pie chart.